Politechnika Wrocławska

Wydział Informatyki i Zarządzania
Kierunek studiów: informatyka

Praca dyplomowa – inżynierska

GRA 2D ZE ZNISZCZALNYM TERENEM ZAIMPLEMENTOWANA W UNITY

Filip Mączko

słowa kluczowe:
lokalna wieloosobowa gra, Unity, destrukcja otoczenia

krótkie streszczenie:
W pracy zaprezentowany jest proces projektowania i implementacja gry, w której możliwe będzie dynamiczne niszczenie terenu rozgrywki, umożliwiającej wieloosobową rozgrywkę na jednym komputerze, testy gry oraz perspektywy jej dalszego rozwoju.

Opiekun pracy dyplomowej			
	Tytuł/stopień naukowy/imię i nazwisko		
Ostateczna ocena za pracę dyplomową			
Przewodniczący Komisji egzaminu dyplomowego			
	Tytuł/stopień naukowy/imię i nazwisko	ocena	podpis

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**

a) kategorii A (akta wieczyste)

b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)

** niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław, rok 2020

STRESZCZENIE

Celem pracy jest zaprojektowanie i implementacja gry akcji, umożliwiającej lokalną wieloosobową rozgrywkę z pełną destrukcją terenu. W pracy przedstawiona została w skrócie obecna sytuacja branży gier wideo oraz to jak kształtują się gusta konsumentów tego medium. Zidentyfikowana została nisza w rynku. Przeanalizowane zostały gry zawierające podobne mechaniki. W pracy przedstawiony został projekt architektury gry bazujący na najnowszych technologiach dostępnych w silniku Unity oraz najciekawsze rozwiązane problemy implementacyjne. Przedstawione zostały wykonane na grze testy wraz z ich wynikami. Zaproponowane zostały możliwości dalszego rozwoju projektu wraz z opisem płynących z nich korzyści. Na końcu pracy porównano otrzymane wyniki z postawionymi wcześniej założeniami.

ABSTRACT

The main goal of this thesis was to create a project of a local multiplayer action game with destructible terrain and implement it. Current state of the industry and consumer preferences were analyzed. Market niche was discovered. Games implementing similar mechanics were analyzed. Design of the game architecture, based on the newest technologies available in Unity engine, was presented, as well as the most interesting technical challenges solved. Test results were presented. Areas of future product development were suggested with short explanations showing benefits of each. At the end the obtained results were compared with the assumptions stated earlier.

SPIS TREŚCI

WPROWADZENIE

WYBÓR PROBLEMU

Praca ma na celu rozwiązanie problemu polegającego na braku gier oferujących wieloosobową rozgrywkę w czasie rzeczywistym z możliwością pełnej destrukcji otoczenia. Na wstępie pracy udzielone zostaną odpowiedzi na trzy kluczowe pytania mające wpływ na wybór problemu oraz tematyki pracy.

Czy opłaca się tworzyć gry?

W dzisiejszych czasach gry wideo stanowią jedną z najbardziej popularnych form rozrywki. Obecnie według raportu opracowanego dla strony `newzoo.com` przez Toma Wijmana [1] w gry gra ponad 2.5 miliarda osób. W dodatku według raportu [1] w ciągu najbliższych trzech lat przybędzie aż około 200 milionów, co daje nam imponującą liczbę 2.7 miliarda graczy w 2023 roku.

Niesamowita popularność tej formy rozrywki przekłada się także na duże dochody tej gałęzi branży IT. Według raportu [1] ten segment rynku w 2019 roku wart był 145.7 miliarda dolarów. Warto zwrócić także uwagę na fakt, że jak wskazuje raport [1] w dobie panującej pandemii koronawirusa Sars Cov 2, rynek gier wideo nie tylko zwiększył swoje dochody (w 2020 roku wartość prognozowana jest na około 159.3 miliarda dolarów), ale także wzrost wartości był znacznie większy niż ubiegłoroczny (wartość w porównaniem z 2019 rokiem wzrośnie o około 14.6 miliarda podczas, gdy różnica pomiędzy wartością z 2018 i 2019 roku wynosiła 7.2 miliarda dolarów). Z przedstawionych danych jednoznacznie wynika, że branża gier wideo jest warta zainteresowania. Dodatkowym czynnikiem motywującym mogą być sukcesy gier pisanych przez niezależne małe zespoły. Za przykład służyć może gra Amoung Us,która według artykułu Michaela Kellyego [2] w samym sierpniu tego roku, była łącznie oglądana na serwisie twitch przez prawie 31 milionów godzin przez co jest uważana za jeden z największych sukcesów branży gier wideo 2020 roku. Została ona stworzona, przez studio developerskie Inner Sloth, które według strony samych autorów [3] składa się jedynie z 4 osób - w tym zaledwie jednego programisty.

Czy lepiej zainteresować się grami wieleoosobowymi czy jednoosobowymi?

Gry możemy podzielić na dwie podstawowe kategorie – gry jednoosobowe i gry wieloosobowe. Według raportu ESA [4] badającego graczy na rynku amerykańskim w 2019 roku, aż 63 procent badanych dorosłych wskazuje, że woli grać z innymi ludźmi. Spędzają oni około 4.8 godziny tygodniowo grając z innymi ludźmi online i około 3.5 godziny grając z innymi ludźmi na tym samym urządzeniu. W związku z tym tworzona w ramach pracy inżynierskiej produkcja, będzie lokalną grą wieloosobową.

Jaki gatunek jest niedostatecznie reprezentowany na rynku gier?

Zgodnie z opinią przedstawicieli Glitch Games, niezależnego studia tworzącego gry od 2012 roku udzieloną w wywiadzie opublikowanym w książce *INDIE GAMES podręcznik niezależnego twórcy gier* [5], jedną z najważniejszych rzeczy jakie należy zrobić przy tworzeniu gry jest zidentyfikowanie niszy w rynku i jej wypełnienie. Nisza taka istnieje w postaci gier akcji ze

zniszczalnym terenem. Gatunek ten niegdyś posiadający wiele różnych reprezentantów aktualnie zdominowany jest przez serię gier Worms, która sprzedała ponad 70 milionów kopii nie oferując przy tym jedynie rozgrywkę w trybie turowym [6].

OPIS PRACY

W ramach pracy zostanie zaprojektowana oraz zaimplementowana lokalna wieloosobowa gra akcji z możliwością pełnej destrukcji otoczenia. Gra ta ma na celu wypełnienie luki zidentyfikowanej w rynku gier wideo we wcześniejszej sekcji. Przedstawione zostaną artefakty procesu projektowania gry oraz architektury je kodu w postaci diagramów klas i aktywności, opisu użytkownika docelowego oraz dokumentu projektu gry. Omówione zostaną najciekawsze rozwiązane problemy implementacyjne oraz metodyka testowania gry, wraz z rezultatami testów. Na końcu pracy zaprezentowane zostaną możliwości dalszego rozwoju gry, a wyniki pracy zostaną podsumowane.

CEL PRACY

Celem pracy jest rozwiązanie przedstawionego problemu, polegającego na braku gier oferujących wieloosobową rozgrywkę w czasie rzeczywistym z możliwością pełnej destrukcji otoczenia, poprzez zaprojektowanie i zaimplementowanie, gry, która wypełni tą lukę w rynku.

ZAKRES PRACY

Zakres prac obejmuje wybór technologii w jakiej powstawać ma gra wraz z jego uzasadnieniem, zaprojektowanie gry poprzez analizę konkurencji, zdefiniowanie grupy docelowych odbiorców, do których gra będzie kierowana, spisanie najważniejszych informacji dotyczących projektu gry w formie dokumentu projektu gry (z ang. Game Design Document) oraz zaprojektowanie architektury systemu pozwalającej na implementację gry wieloosobowej ze zniszczalnym terenem. W pracy zostaną zaprezentowane diagramy klas oraz aktywności obrazujące architekturę systemu. W zakresie prac leży także zaimplementowanie minimalnej wartościowej wersji (z ang. Minimal Viable Game) lokalnej wieloosobowej gry akcji z całkowicie zniszczalnym fizycznym terenem. Zgodnie z artykułem *Making Lean Startup Tactics Work for Games* [7] jest to gra posiadająca absolutne minimum potrzebne do przetestowania głównej idei projektu w przypadku gry platformowej może być to np. jeden poziom. W wypadku gry tworzonej na potrzeby pracy minimalna wartościowa gra sprowadza się do możliwości przeprowadzenia meczy jednego trybu rozgrywki na jednej w pełni zniszczalnej mapie.

1. WYBÓR TECHNOLOGII

Pierwszym zagadnieniem na jakie należy zwrócić uwagę przy tworzeniu gry jest wybór platformy sprzętowej. Jest to niezwykle ważne, ponieważ od wybranej platformy zależy z jakich silników będzie można skorzystać. Wybór odpowiedniego silnika jest również niezwykle istotny. Determinuje on jakich technologi można będzie użyć do tworzenia gry i zmusza do wybrania metody rozwiązania problemów technologicznych w ramach rozwiązań, które są w nim dostępne. Jeśli wybrany silnik oferuje więcej niż jedną możliwość rozwiązania postawionego problemu należy podjąć decyzję dotyczącą, które z nich zamierzamy zaimplementować.

1.1. WYBÓR PLATFORMY SPRZĘTOWEJ

Pisząc grę niezwykle ważny jest wybór odpowiedniej platformy sprzętowej, ponieważ będzie ona miała wpływ na cały dalszy proces, zarówno projektowania jak i implementacji. W swojej książce Richard Hill-Whittal [5], twórca gier z ponad 25 letnim doświadczeniem, zaleca zwłaszcza w początkowej fazie projektu skupienie się na jednej platformie sprzętowej. W przypadku gry będącej przedmiotem tej pracy wybór padł na komputery z systemem `Windows`. Powodem wyboru właśnie tej platformy jest fakt, że budowanie i testowanie gier na tej platformie jest najprostsze do zrealizowania. Wynika to z faktu, że oprogramowanie tworzone jest na tej samej platformie, na której będzie użytkowane, co w znacznym stopniu ułatwia testowanie, które może zostać przeprowadzone nawet bez budowania pliku wykonywalnego. Dodatkowo w przypadku platformy Windows występują najmniejsze różnice pomiędzy zachowaniem prezentowanym w edytorze `Unity`, a tym prezentowanym przez wybudowany plik binarny. Ponadto w przypadku gier na komputery osobiste istnieje możliwość opublikowania ich na platformach takich jak `itch.io` bez konieczności certyfikacji produktu.

1.2. WYBÓR SILNIKA GRY

W przypadku tworzenia gry na komputer osobisty mamy do wyboru wiele silników umożliwiających tworzenie gier komputerowych zarówno takich wymagających programowania, jak i pozwalających na tworzenie gier przez wybór odpowiednich opcji programu – bez konieczności pisania własnych skryptów. Najpopularniejsze silniki według serwisu `gamedesign.org` [8] to:

1. `Unreal Engine` – opisywany w artykule jako [8] jako narzędzie pozwalające tworzyć unikalne doświadczenia w rękach profesjonalisty,
2. `Unity` – według artykułu [8] jest to silnik wysokiej jakości oraz doskonałej funkcjonalności pozwalający tworzyć w zasadzie każdy rodzaj gry,
3. `GameMaker` – silnik pozwalający tworzyć gry bez użycia własnych skryptów,
4. `Godot` - opisywany przez artykuł [8] jako doskonałe narzędzie do tworzenia gier 2D i 3D z otwartym kodem źródłowym,
5. `AppGameKit` – płatny silnik opisywany w artykule [8] jako świetne narzędzie przy tworzeniu gier na platformy mobilne, pozwalające tworzyć gry także na inne platformy w tym na system Windows.

W pracy zgodnie z radą Richarda Hill-Whittala zawartą w jego książce *INDIE GAMES podręcznik niezależnego twórcy gier* [5], zostanie użyty silnik Unity. Jest to silnik umożliwiający implementację gier zarówno 2D, jak i 3D na wiele różnych typów urządzeń, w tym między innymi na komputery osobiste z systemem `Windows` czy urządzenia mobilne zarówno z systemem `Android`, jak i z systemem `iOS`. Posiada on bardzo rozbudowaną i pomocną dokumentację oraz dużą skupia wokół siebie dużą społeczność, aktywnie działającą na forach i blogach. Pozwala on na definiowanie własnych skryptów kontrolujących zachowanie obiektów w języku `C#`.

Do implementacji gry została użyta wersja silnika: `Unity 2020.1.3.1f` – najnowsza dostępna wersja w momencie rozpoczęcia prac nad implementacją gry oraz `C#` 7.3 – najnowsza wersja języka wspierana przez wybraną wersję silnika.

1.3. WYBÓR PODEJŚCIA UŻYTEGO DO WYTWORZENIA ZNISZCZALNEGO TERENU

Chcąc stworzyć grę pozwalającą na dynamiczne niszczenie otoczenia w silniku `Unity` mamy do wyboru dwie opcje pozwalające nam na implementację fizyki:

1. wykorzystanie fizyki zaimplementowanej wewnątrz silnika,
2. zaimplementowanie własnego silnika do obsługi fizyki, opartego na bitmapach reprezentujących fizyczne właściwości każdego piksela terenu.

W przypadku użycia silnika fizycznego wbudowanego w `Unity` komponenty odpowiadające za wykrywanie kolizji należy aktualizować przy każdej zmianie topologi terenu. Jako, że komponenty wykrywające kolizje w przestrzeni 2D zaimplementowane są w oparciu o odcinki, widoczna topologia terenu może nie odpowiadać w pełni topologi wykorzystanej przez fizykę. Dodatkowym problemem występującym przy użyciu wbudowanego silnika fizyczne jest także brak możliwości ustalenia kolejności czynności przez niego wykonywanych. Kolejność wykonywania działań takich jak aktualizacja danych w komponentach odpowiadających za kolizję czy sama wykrywanie kolizji jest stała i nie można jej zmienić. W związku z tym mogą występować problemy takie jak np. wnikanie gracza w teren w momencie kiedy topologia terenu zostanie zmieniona w nieodpowiednim momencie – po wykryciu kolizji.

Implementacja własnej fizyki w środowisku `Unity` jest znacznie trudniejszym rozwiązaniem. Uniemożliwi nam ona skorzystanie z gotowych modułów wykrywania kolizji czy przesuwania obiektów fizycznych. W zamian za to pozwala na pełną kontrolę nad kolejnością wykonywania działań wewnątrz silnika fizycznego, co pomoże zapobiegać błędom oraz w przypadku dobrej implementacji może okazać się znacznie wydajniejszym rozwiązaniem dla tworzonej gry.

Ze względu na przedstawione powyżej argumenty w pracy zaimplementowano własny silnik fizyczny. Opierać się on będzie na bitmapie, w której pojedynczy piksel odpowiadać będzie pojedynczemu pikselowi wyświetlanemu na ekranie.

2. PROJEKT SYSTEMU

Projektowanie systemu to niezwykle ważny proces, które poprawne przeprowadzenie pomaga w określeniu wielu aspektów zarówno biznesowych jak i technicznych projektu. Dobrze zaplanowana architektura kodu pozwala w łatwiejszy sposób pisać dobrej jakości, łatwy w utrzymaniu kod, a przygotowanie dokumentacji gry przed jej wykonaniem pozwala na skupienie się na najważniejszych aspektach tworzonej produkcji.

W tym rozdziale zostanie przedstawiony projekt systemu. Na początku przeprowadzona zostanie analiza gier implementujących podobne mechaniki do tworzonej produkcji. Następnie stworzony zostanie profil użytkownika oraz dokument projektu gry. W dalszej części rozdziału zostanie przedstawiona architektura projektu w postaci diagramów klas oraz aktywności. Zostaną zaprezentowane wireframy interfejsu użytkownika oraz przedstawione zostaną wykonane na ich podstawie elementy menu. Na końcu podane zostaną wymagania dotyczące tworzonej gry.

2.1. ANALIZA KONKURENCYJNYCH ROZWIĄZAŃ

Pierwszym krokiem, który należy podjąć przy tworzeniu projektu - nie tylko informatycznego, jest przyjrzenie się konkurencyjnym rozwiązaniom. Analiza konkurencyjnych rozwiązań pozwala nam ustalić jakie problemy, które możemy napotkać są standardowo rozwiązywane przez naszych konkurentów. Pozwala nam także zaobserwować jakie zmiany mogłyby poprawić lub całkowicie zmienić doświadczenie wynikające z korzystania z produktu, co wykorzystać możemy tworząc własny projekt.

W sekcji tej przeanalizowane zostaną trzy przykłady gier o podobnych mechanikach do gry implementowanej w ramach pracy:

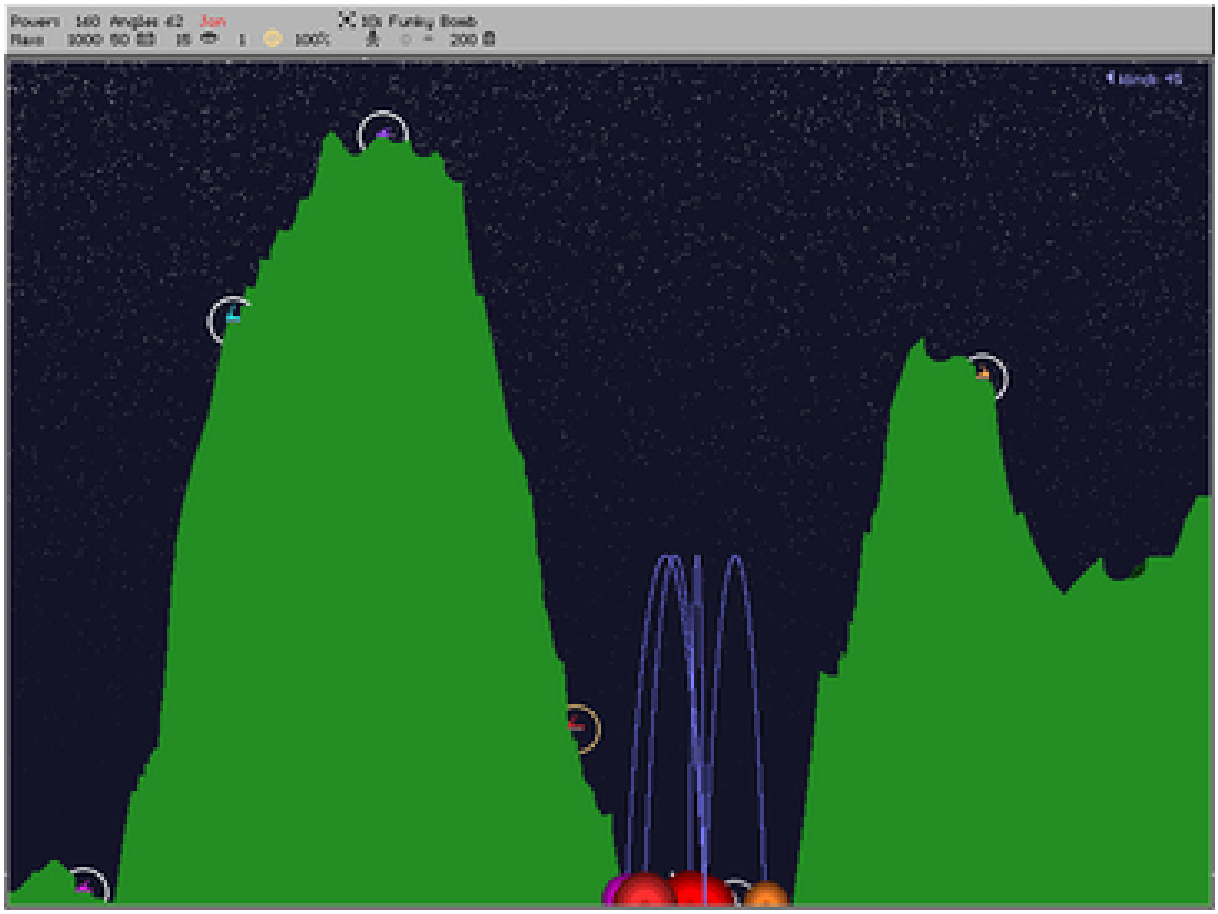
1. `Scorched Eartch` jedna z pierwszych gier implementujących możliwość destrukcji otoczenia,
2. `Liero` - gra najbardziej podobna do tworzonego rozwiązania,
3. `Seria Worms` - seria dominująca rynek gier wieloosobowy gier ze zniszczalnym otoczeniem.

Szczególny nacisk położony zostanie na analizę mechaniki rozgrywki - zwłaszcza silnik zniszczeń terenu produkcji, którego implementacja jest największym problemem technicznym opracowywanej gry.

2.1.1. Scorched Earth

Pierwszą grą, jaką należy przywołać mówiąc o wieloosobowych grach akcji z destrukcją otoczenia jest z pewnością `Scorched Earth`, gra stworzona przez Wendella Hickena i wydana 20 marca 1991 [9], której rozgrywka zaprezentowana jest na rysunku 2.1. Jest to jeden z pierwszych przykładów gry, w której umożliwione było dynamiczne niszczenie terenu oraz poruszanie się po nim.

W grze każdy z graczy, których może być od 2 do 10 licząc w tym także przeciwników sterowanych przez sztuczną inteligencję, zasiada za sterami czołgu. Celem jest zredukowanie zdrowia wszystkich przeciwników do zera w tym samym czasie utrzymując się przy życiu. Aby to zrobić gracze po kolei w systemie turowym zasiadają za sterami swojej postaci, przyjmują



Rys. 2.1. Obraz przedstawiający rozgrywkę gry Scorched Earth.

dogodną pozycję do strzału poprzez przemieszczenie swojego pojazdu i dokonują ostrzału jedną z wielu dostępnych broni. Zredukowanie zdrowia gracza może nastąpić poprzez znajdowanie się w zasięgu eksplozji, przysypanie terenem lub poprzez wypadnięcie poza granice mapy (w tym wypadku zdrowie redukowane jest natychmiastowo do 0). Silnik niszczenia terenu gry nie wspiera latających platform, które stały się popularne w grach implementowanych w późniejszym okresie. W momencie eksplozji piksele, które nie mają kontaktu z podłożem spadają, aż do momentu kontaktu z pozostałą częścią terenu.

2.1.2. Liero

Następnym tytułem, który warto poruszyć jest gra `Liero` stworzona przez Joosa Riekkinen i wydana w 1998 roku [10], której rozgrywka została zaprezentowana na obrazku 2.2.

W grze tej dwóch graczy rywalizuje ze sobą próbują wyeliminować się nawzajem poprzez zredukowanie zdrowia przeciwnika do zera. Wyjątkową cechą tej produkcji jest fakt, że wszystkie gra odbywa się w czasie rzeczywistym. Każdy z graczy może poruszać się i niszczyć otoczenie oraz używać linki do sprawniejszego przemieszczania się. Widok w grze realizowany jest przy pomocy tak podzielonego ekranu, co umożliwia każdemu z graczy zorientowanie się, gdzie się znajduje. Silnik destrukcji otoczenia tej gry w przeciwieństwie do prezentowanego wcześniej `Scorched Earth` pozwala na istnienie wiszących platform. W silniku zaimplementowana jest również możliwość tworzenia niezniszczalnych obiektów w świecie gry.

2.1.3. Seria Worms

Mówiąc o wieloosobowych grach akcji z destrukcją otoczenia, nie można nie powiedzieć o serii `Worms`, która jest dominującym przedstawicielem tego gatunku. Pierwsza gra z serii `Worms` została wydana w 1995 roku i dotychczas ukazało się ponad 20 tytułów wchodzących w skład serii, oferujących rozgrywkę zarówno w przestrzeni 2D jak i 3D [11].

W tym podrozdziale chciałbym przeanalizować grę `Worms Armageddon`, której rozgrywka przedstawiona jest na rysunku 2.3. Według magazynu `pcgamesn.com` [12] jest dalej uważana za jedną z najlepszych odsłon serii mimo, że została wydana w 1999 roku. Według artykułu [8] jest to doświadczeniem skupiające się najbardziej na bazowej mechanice do czego w ostatnich odsłonach serii twórcy starali się wracać rezygnując np. z mechanik cieczy.

W grze tej każdy gracz przejmuje kontrolę nad drużyną składającą się z kilku robaków. Jego celem jest wyeliminowanie wszystkich robaków przeciwnika poprzez zredukowanie ich zdrowia do zera, bądź wtrącenie ich do wody znajdującej się u dołu planszy, zanim przeciwnikowi uda się dokonać tego samego. Gracze po kolei w trybie turowym przejmują kontrolę nad jednym robakiem ze swojej drużyny, którym mogą się poruszać oraz dokonywać ostrzału z najróżniejszych rodzajów broni.

Silnik niszczenia fizyki w grze jest bardzo podobny do silnika gry `Liero`. Tu również wspierane jest występowanie wiszących platform oraz niezniszczalne obiekty, które mogą przyjmować najróżniejsze kształty.

2.1.4. Podsumowanie analizy konkurencyjnych rozwiązań

Na podstawie przedstawionych rozwiązań widzimy, że standardem w wieloosobowych grach akcji ze zniszczalnym terenem stało się rozwiązanie umożliwiające powstawanie lewitujących fragmentów terenu stosowane zarówno w grach z serii `Worms` jak i w `Liero`.

Na rysunkach prezentujących rozgrywkę 2.1 2.2 2.3 możemy zauważyć, że wszystkie produkcje oferujące możliwość dynamicznego niszczenia terenu zdają się używać podejścia opartego na bitmapach prezentowanego we wcześniejszej części pracy.

Warto także zwrócić uwagę na fakt, że we wszystkich prezentowanych grach poza `Liero` istnieje możliwość prowadzenia rozgrywki wieloosobowej w gronie znajomych, którego wielkość w pewnych ramach może być wybierana przez użytkownika - wormy oferują rozgrywkę do 6 osób na raz, a `Scorched Earth` aż do 10.

2.2. OPIS UŻYTKOWNIKA

Wiedząc już jak wyglądają rozwiązania oferowane przez konkurencyjne gry możemy przystąpić do identyfikacji potencjalnego użytkownika naszej gry. Jest to niezwykle ważne, ponieważ daje nam to istotny punk odniesienia przy podejmowaniu dalszych decyzji projektowych. Tylko wiedząc do kogo kierowana jest nasza gra możemy stworzyć doświadczenie, z którego użytkownik będzie zadowolony.

Potencjalny użytkownik tworzonej przez nas gry jest osobą, która lubi gry akcji oraz destrukcję otoczenia. Preferuje on granie z innymi ludźmi niż jednoosobowe doświadczenia. Chciałby móc grać z przyjaciółmi w grę, w której mogą rywalizować między sobą w czasie rzeczywistym, bez potrzeby przekazywania sobie kontrolera jak często ma to miejsce w systemie turowym.

Wiedząc jakie cechy posiada potencjalny użytkownik naszej gry, dzięki raportowi przygotowanemu przez ESA możemy zidentyfikować najważniejsze jego cechy takie jak płeć czy wiek. Gry akcji, według raportu *2019 Essential Facts About the Computer and Video Game Industry* [4], są ulubioną formą rozgrywki mężczyzn w wieku 18–34 lata - w grupie tej, aż 83 procent badanych

wskazuje, że jest to najczęściej ogrywany przez nich gatunek. Grupa ta ma także większy odsetek osób preferujących granie ze znajomymi wynoszący aż 66 procent w porównaniu do 63 procent wśród dorosłych graczy [4]. W związku z tym to właśnie młodzi mężczyźni w wieku od 18 do 34 lat stanowić będą bazę docelowej grupy odbiorców do której kierowana będzie gra stworzona w ramach pracy inżynierskiej.

2.2.1. Wymagania

Znając profil użytkownika dla, którego ma być tworzona gra możemy przejść do definiowania wymagań. Zgodnie z opisem Marka Grochowskiego [13] zdefiniowanie tego jak system ma się zachowywać poprzez wymagania funkcjonalne oraz to jakie standardy ma spełniać poprzez wymagania niefunkcjonalne.

W dalszej części rozdziału przedstawione zostały wymagania funkcjonalne oraz niefunkcjonalne dla gry tworzonej w ramach tej pracy inżynierskiej.

2.2.1.1. Wymagania funkcjonalne

1. możliwość usuwania części terenu w momencie wybuchu
2. możliwość poruszania się gracza po zniszczonym terenie
3. możliwość poruszania się gracza w powietrzu
4. możliwość równoczesnego grania dla od 2 do 4 graczy

2.2.1.2. Wymagania niefunkcjonalne

1. wsparcie dla komputerów z systemem Windows 10.
2. płynne działanie (średnia przekraczająca 30 klatek na sekundę) na komputerze ze średniej półki cenowej (minimum 4 wątki o taktowaniu na poziomie 2 - 3 GHz).
3. wsparcie dla różnego rodzaju kontrolerów (kontrolery od konsoli `XBox`, `PlayStation` i kontrolery third party).
4. szybkie wczytywanie - mapy gry powinna wczytywać się mniej niż 15 sekund. Granica 15 sekund wybrana została na podstawie artykułu Robeterta Millera *Response time in man-computer conversational transactions* [14], gdzie wartość 15 sekund została postawiona jako czas, który użytkownik jest w stanie odczekać w przypadku ładowania danych.
5. czytelność - gracz w łatwy sposób może zorientować się w jakim miejscu planszy się znajduje oraz w którą stronę poleci wystrzelony przez niego pocisk

2.3. DOKUMENT PROJEKTU GRY

Ze względu na to, że projektowanie gry w znaczący sposób różni się od projektowania oprogramowania stricte biznesowego, również metody projektowania takiego rodzaju oprogramowania są różne. Historyjki użytkownika czy diagramy przypadków użycia są stosowane bardzo sporadycznie. Zamiast nich używany jest dokument projektu gry (z ang. Game Design Document) opisujący najważniejsze zagadnienia projektu. W zależności od gatunku i wielkości projektu zawartość takiego dokumentu może się różnić. Na potrzeby pracy użyty zostanie format zaproponowany przez `makeschool.com` na kursie *IOS GAMES SUMMER ACADEMY 2017* [15]. Został on wybrany ze względu na fakt, że dostosowany jest on do potrzeb małego projektu, którego celem końcowym jest utworzenie gry spełniającej założenia minimalnej wartościowej wersji gry. Bazując na strukturze dokumentu, zaproponowanej w kursie [15], powinien on posiadać następujące rozdziały:

1. Projekt gry – w tym punkcie zostanie opisany cel gry, jej główna mechanika oraz to jak projektowane będą poziomy

2. Strona techniczna – w tym punkcie zostanie opisane jakie najważniejsze sceny zawierać będzie projekt, jaki urządzenia wejścia będą obsługiwane oraz w jaki sposób będzie się to działo, przedstawiony zostanie też zarys najważniejszych klas.
3. Kamienie milowe – w tym punkcie zostaną przedstawione kamienie milowe projektu.

2.3.1. Projekt gry

2.3.1.1. Cel gry

Celem projektowanej gry jest wygranie trybu gry powszechnie znanego jako "deathmatch". Każdy z graczy, których na mapie może być jednocześnie od 2 do 4 kieruje jedną postacią i ma za zadanie wyeliminować przeciwników przy pomocy eksplozji powstających w wyniku kolizji pocisków z innymi pociskami, graczami lub podłożem. Jednocześnie samemu nie zostając musi on unikać eksplozji oraz wypadnięcia poza mapę. Ostatni żywy gracz wygrywa.

2.3.1.2. Mechanika gry

Do dyspozycji graczy zostaje oddana w pełni zniszczalna mapa, po której mogą poruszać się w czasie rzeczywistym biegając, wspinając się po wzniesieniach terenu i skacząc. Każdy z graczy ma możliwość wystrzeliwania pocisków, niszczących otoczenie. W momencie kolizji wystrzelonego pocisku z graczem, innym pociskiem bądź terenem następuje eksplozja, która może zabić gracza jeśli ten znajduje się za blisko. Niszczy ona także teren znajdujący się w jej zasięgu.

2.3.1.3. Projekt poziomów

W grze dostępna będzie jedna mapa składająca się z głównej wyspy oraz dwóch latających wysp. Na początku poziomu gracze będą umiejscawiani w różnych predefiniowanych częściach mapy.

2.3.2. Strona techniczna

2.3.2.1. Sceny

Rozgrywka przeprowadzana będzie na jednej scenie zawierającą mapę oraz wszystkich graczy. Poza sceną rozgrywki w grze znajdą się również scena menu głównego, z której przejść będzie można do wytłumaczenia zasad i sterowania oraz menu odpowiadającego za dołączanie graczy do rozgrywki, menu pozwalające graczom dołączać do rozgrywki oraz scena tłumacząca podstawowe założenia rozgrywki oraz sterowanie.

2.3.2.2. Metoda sterowania

Każdy z graczy będzie posiadał własny kontroler z dwoma gałkami analogowymi. Prawa gałka służyć będzie do celowania, położenie celownika będzie odpowiadać jej wychyleniu. Lewa gałka odpowiadać będzie za kontrolowanie kierunku ruchu postaci. Prędkość gracza zależeć będzie od tego jak bardzo zostanie ona wychylona w danym kierunku. Przycisk R1, znajdujący się się pod prawym palcem wskazującym gracza, służyć będzie do oddawania strzałów, a L1, znajdujący się pod lewym palcem wskazującym gracza, do skakania. W ten sposób wszystkie akcje będą możliwe do wykonania bez odrywania kciuków od gałek analogowych co umożliwi ciągłość ruchu i celowania.

2.3.2.3. Najważniejsze klasy

Architektura gry zostanie oparta o paradygmat ECS [16]. Działaniem wszystkich systemów na scenie zarządzać będzie klasa `LoopSolver` zawierająca listę wszystkich systemów odpowia-

dających za logikę rozgrywki. Do przechowywania danych na obiektów użyte zostaną klasy komponentów. Każda z nich będzie przechowywać jedynie dane niezbędne do działania pewnego aspektu systemu przez co będą one wewnętrznie zależne np. komponent odpowiadający za grawitację będzie zawierał jedynie informacje o tym jakie przyśpieszenie powinno działać na obiekt, dane z niego będą jednak wykorzystywane przy wyliczaniu pozycji, do których wykorzystane będą również dane z komponentu odpowiedzialnego za ruch.

2.3.3. Kamienie milowe

Kamienie milowe to według wpisu w Encyklopedii Zarządzania na ten temat zredagowanego przez Agnieszkę Wolf jednorazowe ważne zdarzenia dotyczące konkretnego projektu.

W ramach pracy nad grą, będącą przedmiotem tej pracy wyszczególnione zostały następujące kamienie milowe, po zrealizowaniu których prace nad minimalną wartościowej grą będącą przedmiotem tej pracy inżynierskiej będzie można uznać za skończone.

1. Zakończenie procesu projektowania architektury gry.
2. Zakończenie prac nad systemem umożliwiającym usuwania fragmentów terenu.
3. Zakończenie prac nad systemem umożliwiającym wykrywanie kolizji z terenem.
4. Zakończenie prac nad systemem umożliwiającym poruszanie się obiektów w powietrzu.
5. Zakończenie prac nad systemem umożliwiającym wykrywanie kolizji pomiędzy pociskami i graczami.
6. Zakończenie prac nad systemem poruszanie się gracza.
7. Zakończenie procesu testowania gry i analizy wyników testów.

2.4. ARCHITEKTURA SYSTEMU

Zaprezentowana architektura bazowana jest na nowym paradygmacie ECS (Entity Component System), zaproponowanym przez Unity. Architektura ta, według artykułu, w którym została zaprezentowana [16], stanowi centrum nowego stosu technologicznego zorientowanego na danych – DOTS (Data Oriented Technological Stock) tworzonego przez Unity. Zgodnie z założeniami ECS skrypty w grze zostały podzielone na systemy i komponenty. Komponenty odpowiadają za przechowywanie danych i mogą być podpinane do wielu obiektów gry stosowanych jako jednostki (z ang. entity) w opracowywanej grze. Mogą być one wzajemnie zależne w tym także wzajemnie się zawierać. Drugim typem skryptów są są systemy, które pobierają informacje o wszystkich aktywnych komponentach danego rodzaju i wykonują na nich określone operacje. Działania w każdym systemie wykonują się co klatkę i są one wzajemnie niezależne, jednak mogą one operować na danych z tych samych komponentów przez co niezwykle ważna jest kolejność ich wykonania.

Użycie paradygmatu ECS w projekcie w perspektywie czasu pozwoli na bezproblemowe przeniesienie gry do DOTS (Data-Oriented Technological Stack), kiedy ten zostanie już ukończony.

2.4.1. Diagram klas

Diagram ten jest zgodnie z artykułem pani Beaty Frączek na jego temat [17] jest jednym z najważniejszych diagramów UML. Pozwala nam on na przejrzyste ukazanie struktury systemu poprzez pokazanie w graficzny sposób statycznych zależności między klasami. Przygotowany diagram prezentujący projekt klas systemu zgodny z paradygmatem ECS został przedstawiony na rysunku 2.4. Na diagramie wyszczególnione zostały osobne pakiety zawierające komponenty i systemy. Jednostki (z ang. entities) z prezentowanej architekturze przyjmują formę obiektów gry, przez co nieobecne są na diagramie.

2.4.2. Diagram aktywności

Przy symulowaniu na bieżąco całkowicie zniszczalnego terenu ważne jest zapewnienie odpowiedniej kolejności wykonywania poszczególnych działań, ponieważ niektóre z operacji zależą od wyniku działania poprzednich. Dla przykładu można podać, że system odpowiedzialny za poruszania się gracza po terenie potrzebuje informacji o stanie terenu z danej klatki inaczej mógłby np. przesunąć gracza w powietrze, gdzie klatkę temu znajdował się jeszcze teren.

W związku z tym ważne jest, aby już na etapie projektowania przemyśleć w jakiej kolejność powinny być wykonywane akcje. W celu zobrazowania tej kolejności w pracy przygotowany został diagram aktywności, polecany do tego celu przez Iwonę Rostowską w artykule *Diagram aktywności* na stronie `zwinna.pl` [18]. Diagram został przedstawiony na rysunku 2.5.

2.4.3. Wireframe interfejsu

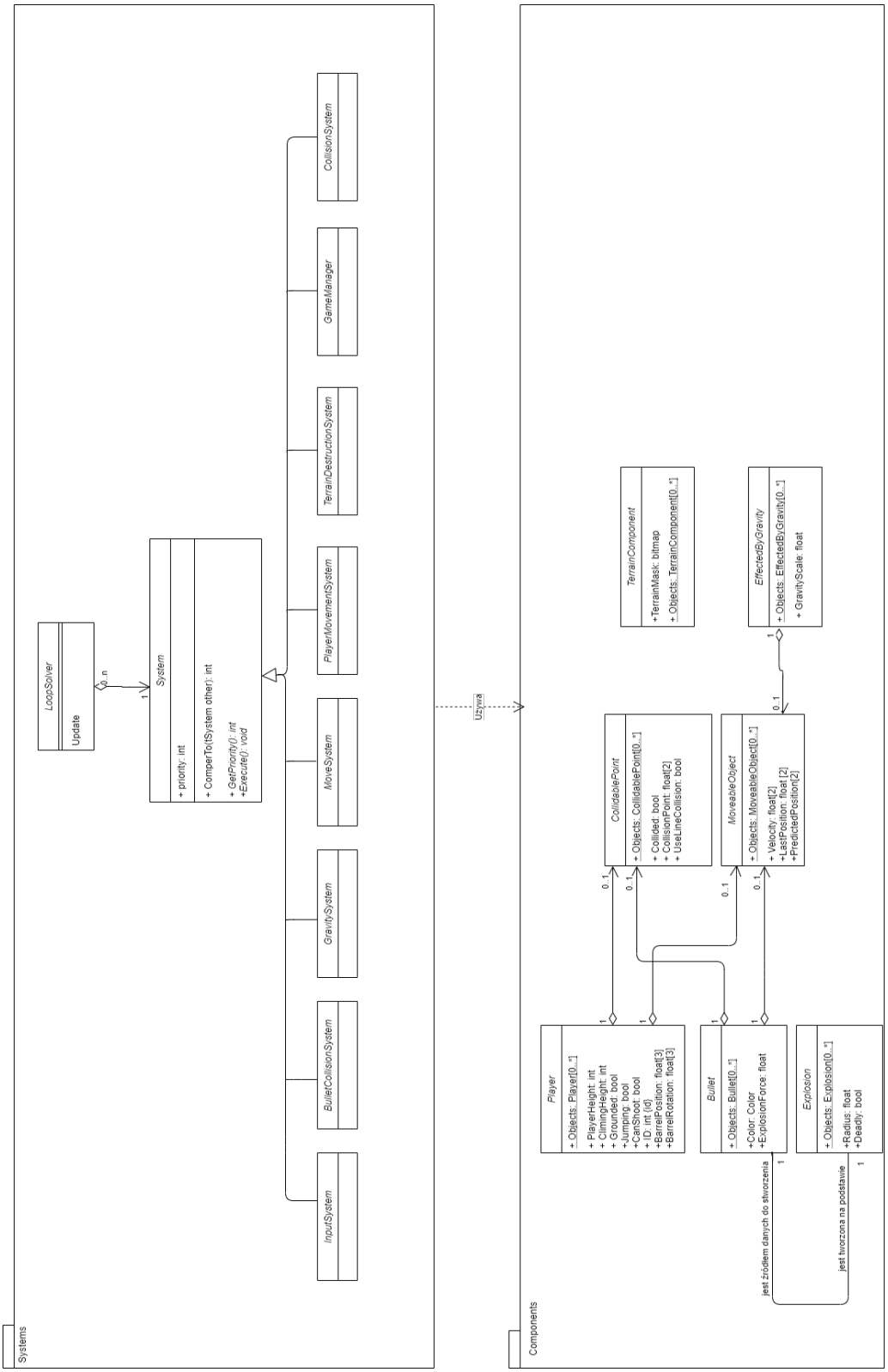
Według `justinmind.com` [19] wireframy są szkieletem interfejsu użytkownika, który zabierając stosunkowo najmniej czasu pozwala na przetestowanie tworzonego interfejsu przed jego implementacją. Jest to niezwykle ważne, ponieważ tworząc interfejs graficzny bez prototypu może łatwo przeoczyć funkcjonalności lub o nich zapomnieć, a dokonywanie zmian w zaimplementowanym już projekcie jest znacznie bardziej kosztowne niż zmiana w fazie projektowania.

Do wykonania wireframów dla projektowanej gry został użyty program `Axure`. Wireframy wykonano dla 3 statycznych ekranów menu oraz dynamicznego panelu prezentującego kto wygrał mecz. Wireframy wraz z zrzutami ekranu przedstawiającymi interfejs w minimalnej wartościowej grze:

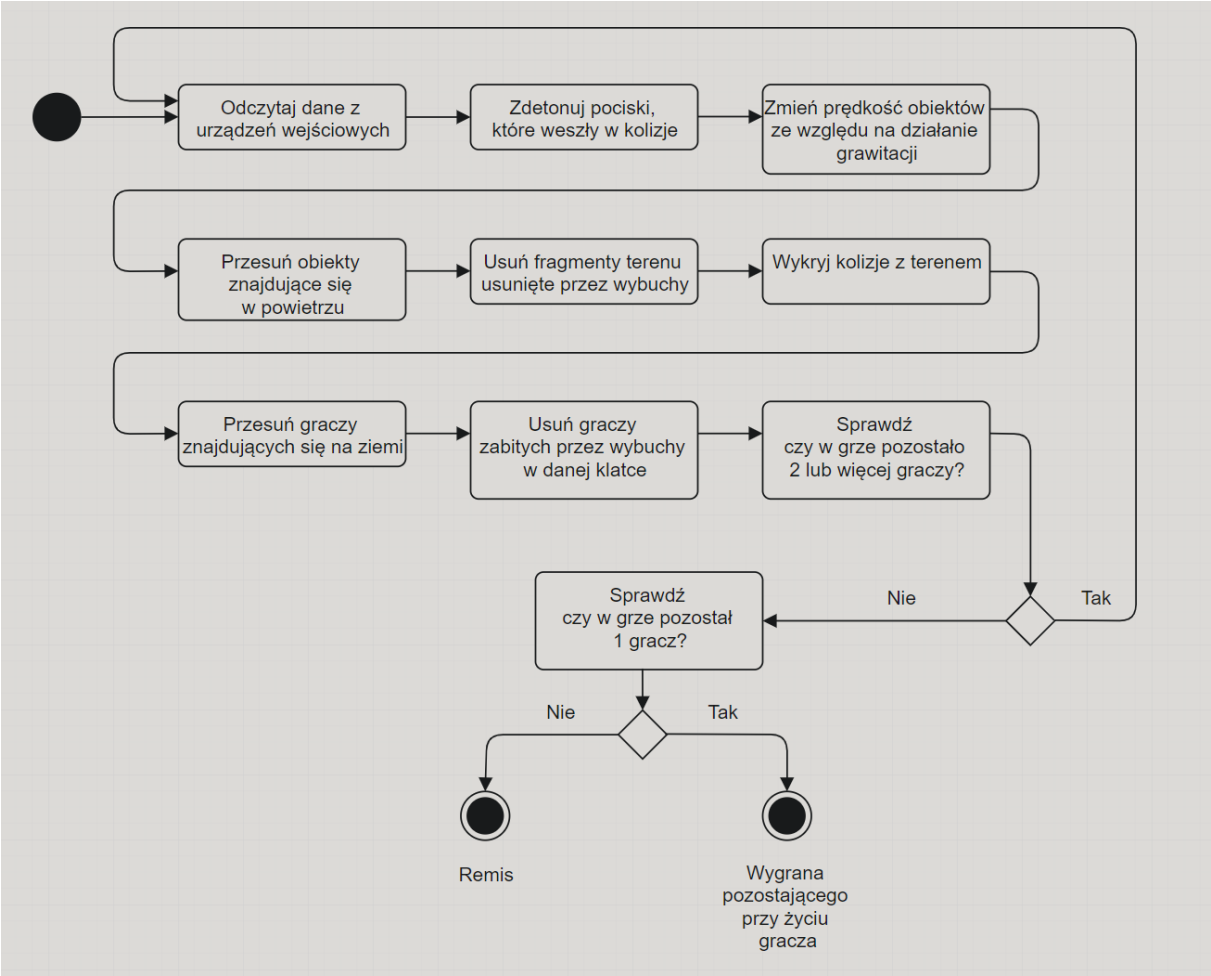
1. menu główne – wireframe przedstawiony na rysunku 2.6, zrzut ekranu na rysunku 2.7,
2. menu rozgrywki pozwalającemu na dołączanie graczy – wireframe przedstawiono na rysunku 2.8, zrzut ekranu na rysunku 2.9,
3. ekranu objaśniającego sterowanie i cele rozgrywki – wireframe przedstawiono na rysunku 2.10, zrzut ekranu na rysunku 2.11,
4. ekranu pojawiającego się, gdy gra zostanie zakończona i któryś z graczy wygra bądź nastąpi remis – wireframe przedstawiono na rysunku 2.12, zrzut ekranu na rysunku 2.13.

Rys. 2.2. Obraz przedstawiający rozgrywkę gry Liero.

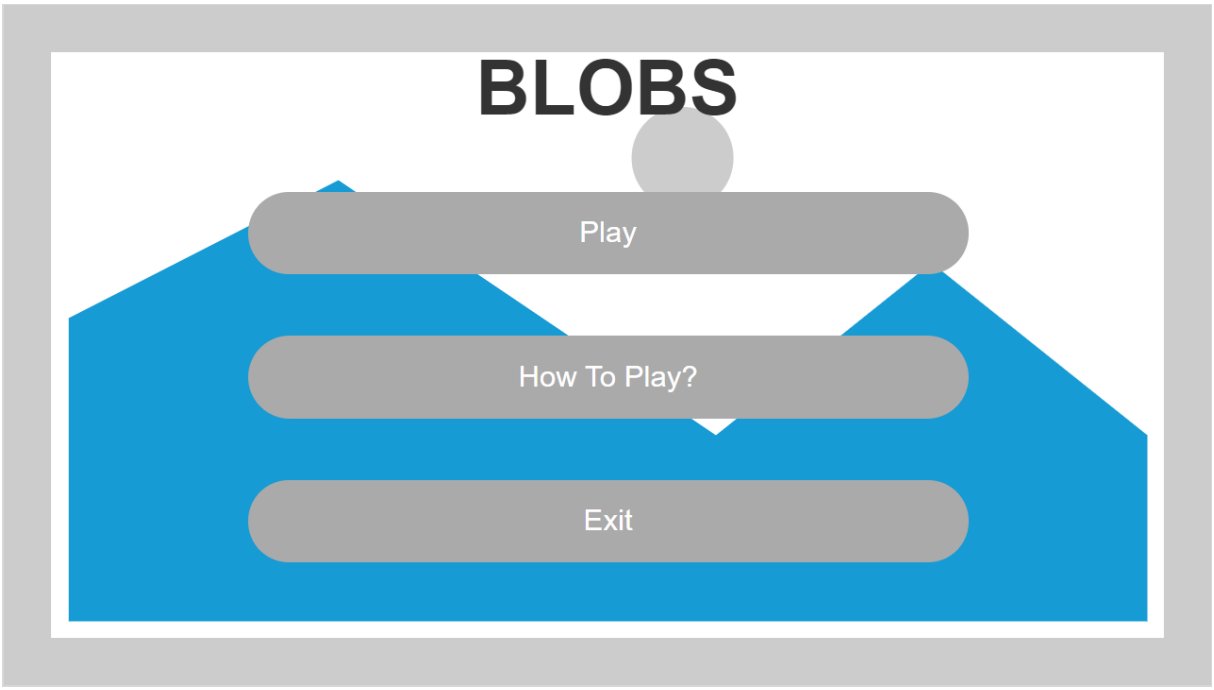
Rys. 2.3. Obraz przedstawiający rozgrywkę z gry Worms Armageddon.



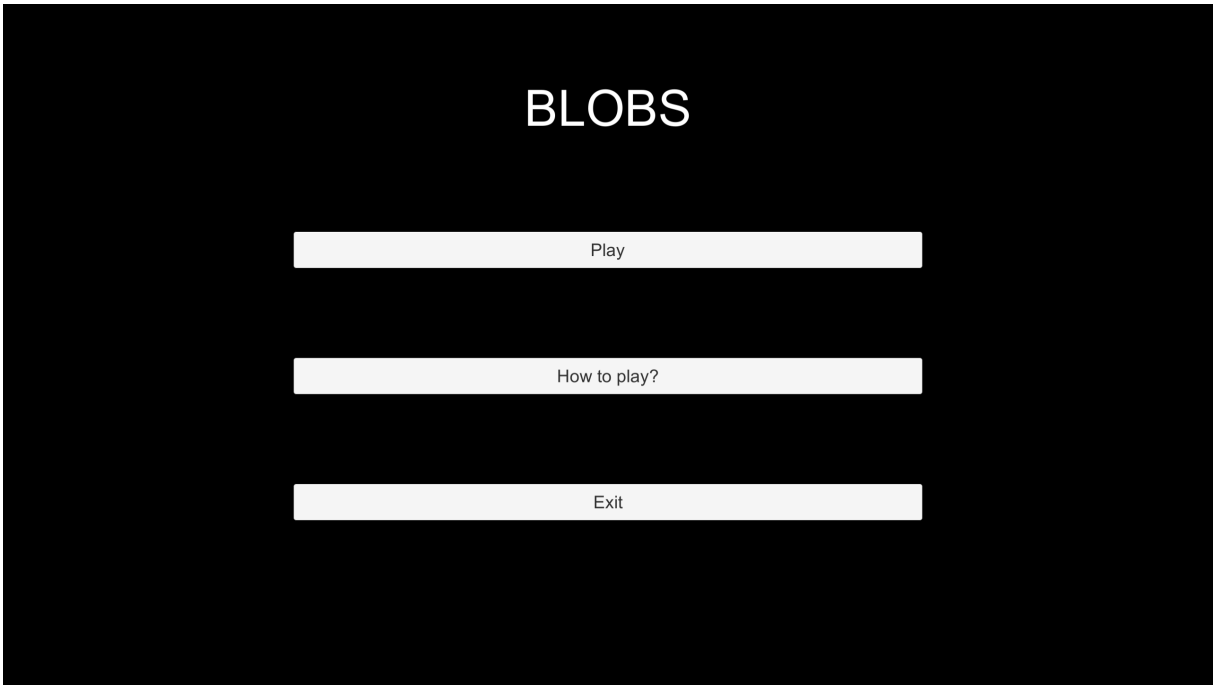
Rys. 2.4. Diagram klas obrazujący zależności pomiędzy klasami w projektowanym systemie.



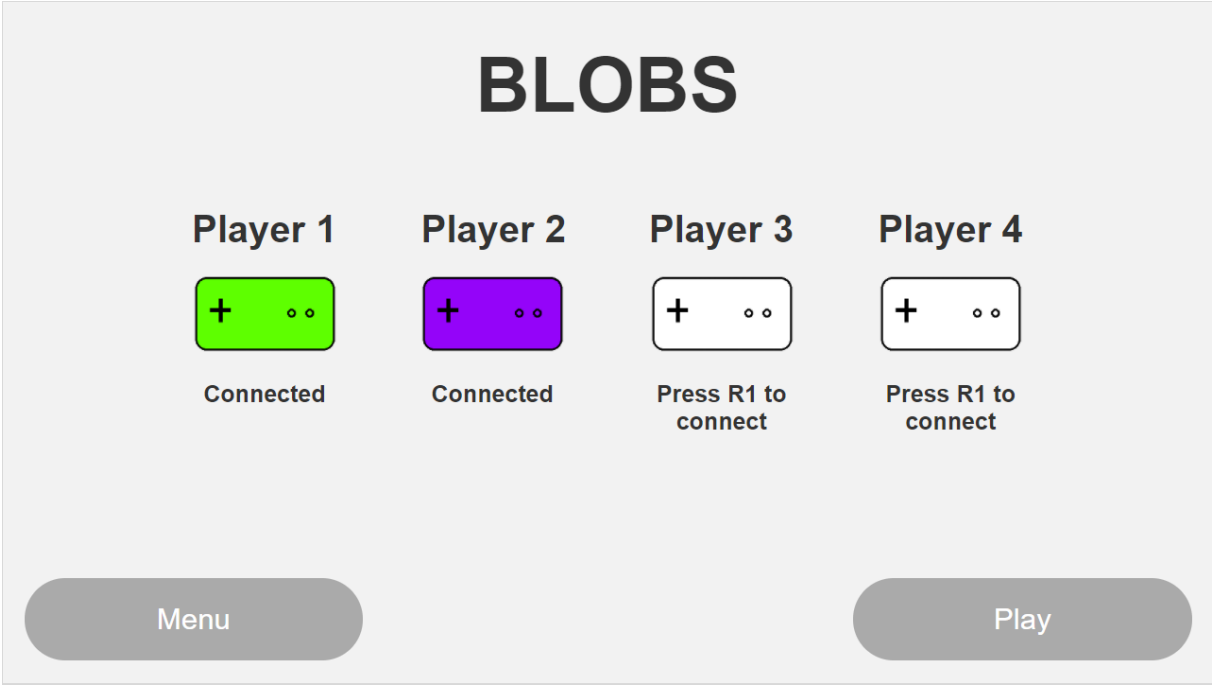
Rys. 2.5. Diagram aktywności obrazujący kolejność wykonywania czynności w grze.



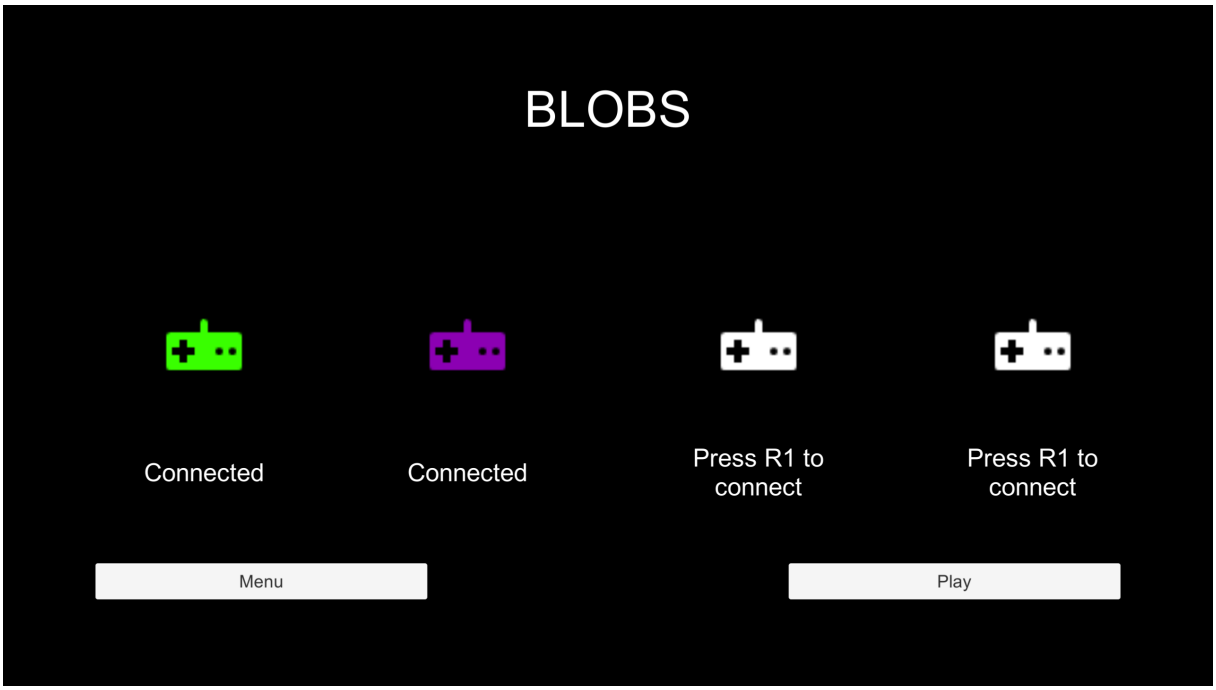
Rys. 2.6. Obraz przedstawiający wireframe głównego menu gry.



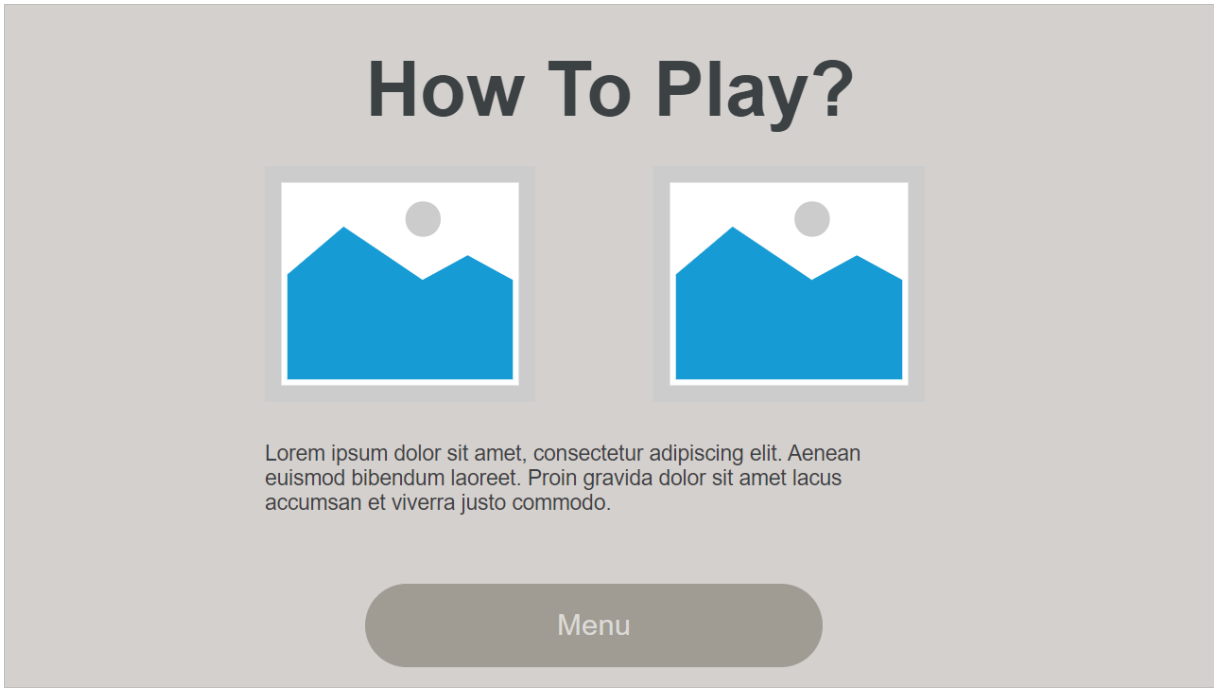
Rys. 2.7. Zrzut ekranu z zaimplementowanej minimalnej wartościowej gry przedstawiający menu główne.



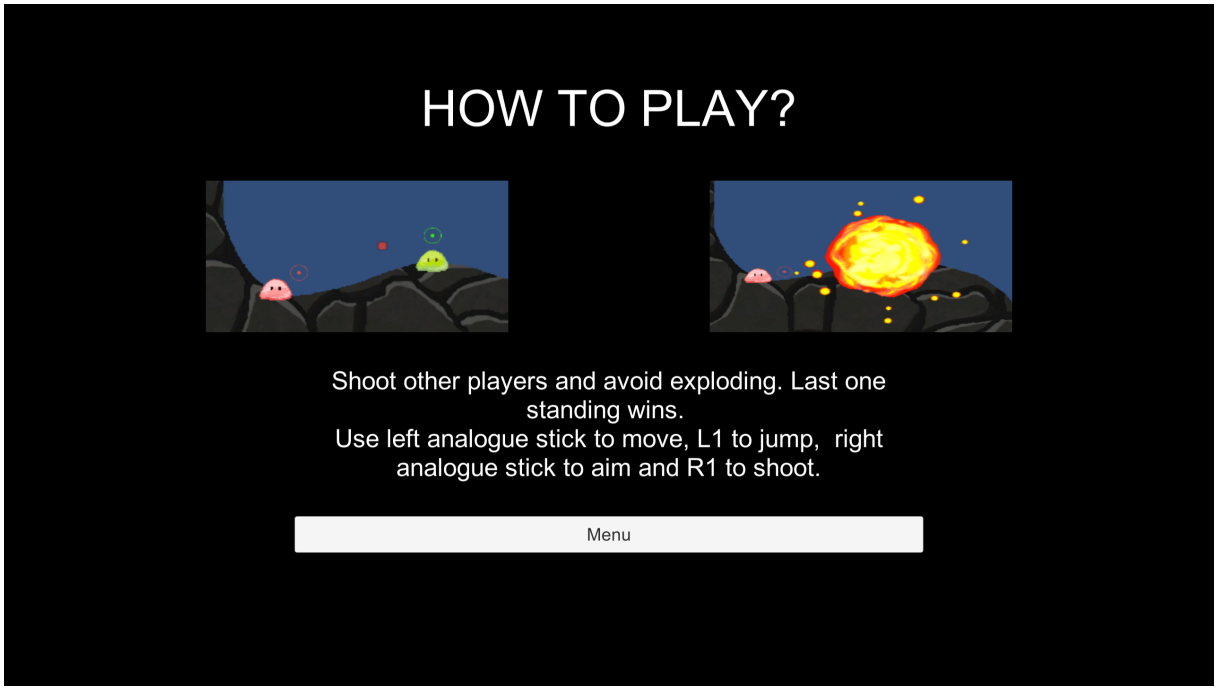
Rys. 2.8. Obraz przedstawiający wireframe ekranu umożliwiającego dołączanie graczy do rozgrywki.



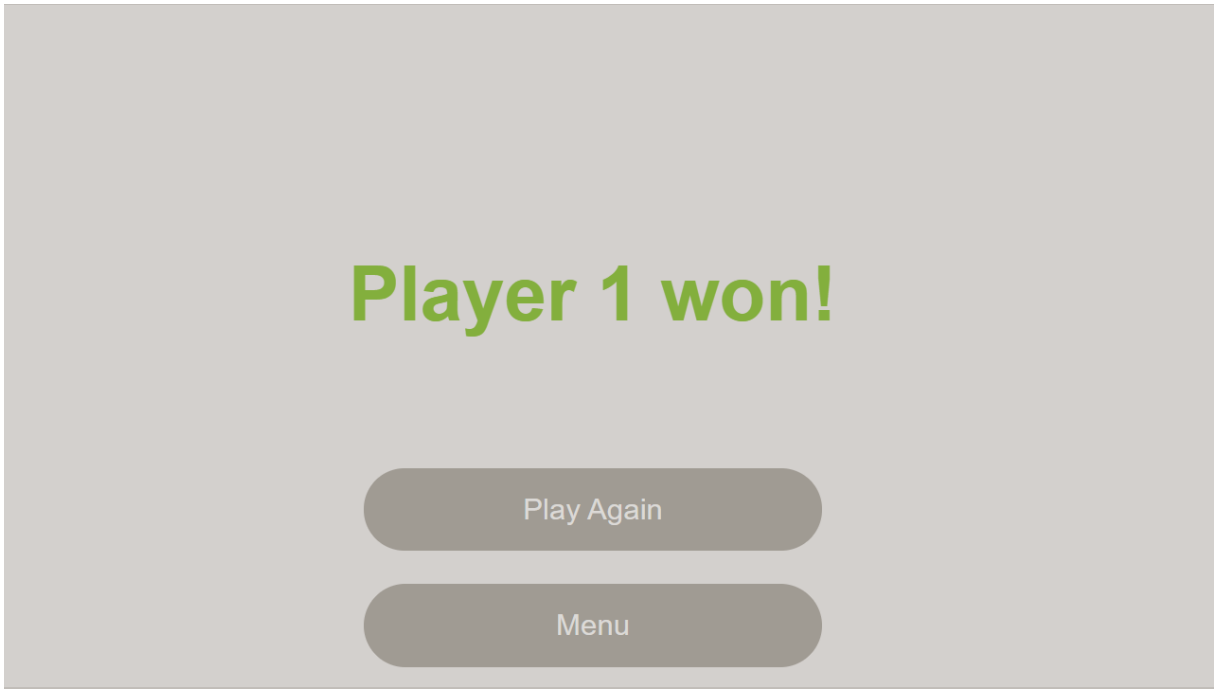
Rys. 2.9. Zrzut ekranu umożliwiającego dołączenie graczy z zaimplementowanej minimalnej wartościowej gry.



Rys. 2.10. Obraz przedstawiający wireframe ekranu tłumaczącego sterowanie i podstawowe założenia rozgrywki.



Rys. 2.11. Zrzut ekranu przedstawiający sterowanie i podstawowe założenia rozgrywki z zaimplementowanej minimalnej wartościowej gry.



Rys. 2.12. Obraz przedstawiający wireframe ekranu pokazującego wygraną gracza.

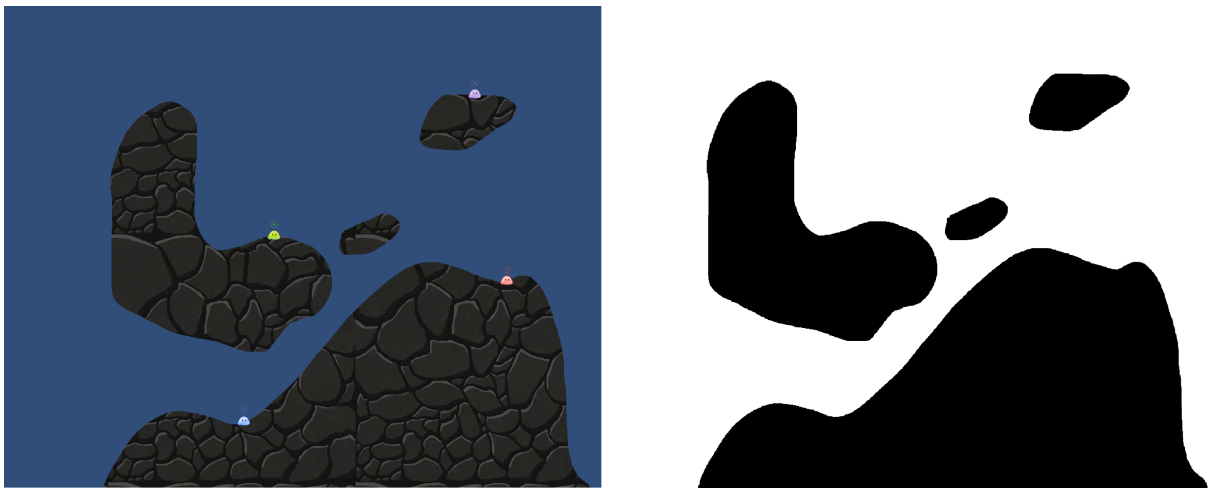
Rys. 2.13. Zrzut ekranu przedstawiający ekran mówiący o zwycięstwie gracza z zaimplementowanej minimalnej wartościowej gry.

3. ROZWIĄZANE PROBLEMY IMPLEMENTACYJNE

W trakcie implementowania gry, będącej przedmiotem pracy inżynierskiej napotkane zostało wiele problemów związanych zarówno z tworzonym silnikiem odpowiadającym za fizykę umożliwiającą destrukcję otoczenia jak i z zagadnieniami technicznymi specyficznymi dla wybranego silnika - `Unity`. Najciekawsze z nich wraz z omówieniem ich rozwiązania zostały zaprezentowane w tym rozdziale.

3.0.1. Wizualna reprezentacja zniszczeń terenu.

W tworzonej grze niezwykle ważne jest, aby gracze w każdym momencie wiedzieli, które fragmenty terenu zostały zniszczone, a po których dalej można się poruszać. W celu rozwiązania tego problemu możemy wykorzystać tak zwane Sprite Maski. Zgodnie z opisem zawartym w *Introduction To Sprite Mask* [20] pozwalają one na rysowanie, tylko części obrazka znajdującego się tylko wewnątrz lub na zewnątrz maski. Dzięki temu możemy użyć bitmapy, której wykorzystujemy do obliczeń związanych z fizyką terenu, również do jego wyświetlania, co pozwala nam unikać rozbieżności pomiędzy stanem terenu widzianym przez gracza, a stanem widzianym przez system fizyki, co zaprezentowane zostało na rysunku 3.1.



Rys. 3.1. Obraz pokazujący zestawienie maski wykorzystywanej w grze z mapą w grze.

3.0.2. Wykonywanie systemów w określonej kolejności.

W dokumentacji `Unity` [21] przedstawiony jest domyślna kolejność wywoływania się funkcji. Na schemacie widzimy, że co klatkę wywoływana jest funkcja `Update`, a funkcje związane z wewnętrzną implementacją fizyki wywoływane są w pętli n razy przed wywołaniem logiki gry. Należy zwrócić uwagę, że funkcja `FixedUpdate`, w której domyślnie przelicza się fizykę może wykonać się wiele razy na klatkę w zależności od tego ile zajęło jej wygenerowanie. W przypadku zaproponowanej architektury opartej na systemach, powinny one wywoływać się

one raz na klatkę w przedstawionej uprzednio na diagramie przepływu predefiniowanej kolejności. Domyślnym sposobem tworzenia kodu jest zapisanie logiki, która ma zostać wykonana co klatkę w funkcji `Update` skryptu dziedziczącego po klasie `MonoBehaviour`. Niestety to rozwiązanie niesie za sobą kilka problemów. Według artykułu *1000 Update calls* napisanego przez Valentina Simonova [22] najważniejsze z nich to brak pewności jak wykonywane są funkcje wbudowane w silnik, niejasność co do kolejności wykonywania funkcji `Update` przez różne skrypty oraz brak poprawnej współpracy z intellisense. Największą wadą z punktu widzenia tworzonej gry jest utrudniona kontrola nad kolejnością wykonywania funkcji Update przez różne skrypty. Ustawienie kolejności wykonania skryptów jest możliwe przez odpowiednie menu (menu: Edit > Project Settings > Script Execution Order). Rozwiązanie to jednak nie jest wygodne ze względu na konieczność ręcznej zmiany kolejności po dodaniu kolejnego systemu. W celu rozwiązania tego problemu została utworzona klasa `LoopSolver` dająca nam pełną kontrolę na kolejnością wykonywania systemów. Jest ona jedyną klasą w całej aplikacji, która zarządza kolejnością wykonywania systemów. Do sprawnego działania utworzonej klasy potrzebna jest nam opcja pozwalająca na wykrycie wszystkich systemów i uporządkowanie ich. W tym celu w języku C# w wersji 7.3 możemy posłużyć się interfejsem lub klasą abstrakcyjną. Interfejsy nie posiadają niestety w najnowszej wersji języka używanej w `Unity` zgodnie z informacjami zawartymi w książce *C# a quick syntax reference* [23] możliwości implementacji ciała funkcji. Ze względu na to oraz fakt, że sortowanie systemów jest wymagane do poprawnego działania klasy `LoopSolver` w pracy została zaimplementowana klasa abstrakcyjna `System`, po której dziedziczą wszystkie systemy.

3.0.3. Usuwanie terenu przy wybuchu

Jedną z najważniejszych opcji w każdej grze artyleryjskiej jest usuwanie terenu na skutek wybuchu. Sprowadza się to wykonania dwóch czynności: wykrycia wybuchu oraz usunięcia pikseli znajdujących się w jego zasięgu z maski używanej zarówno do wizualnej reprezentacji terenu jak i do wyliczania fizyki. W tym fragmencie skupimy się na drugim z wymienionych aspektów.

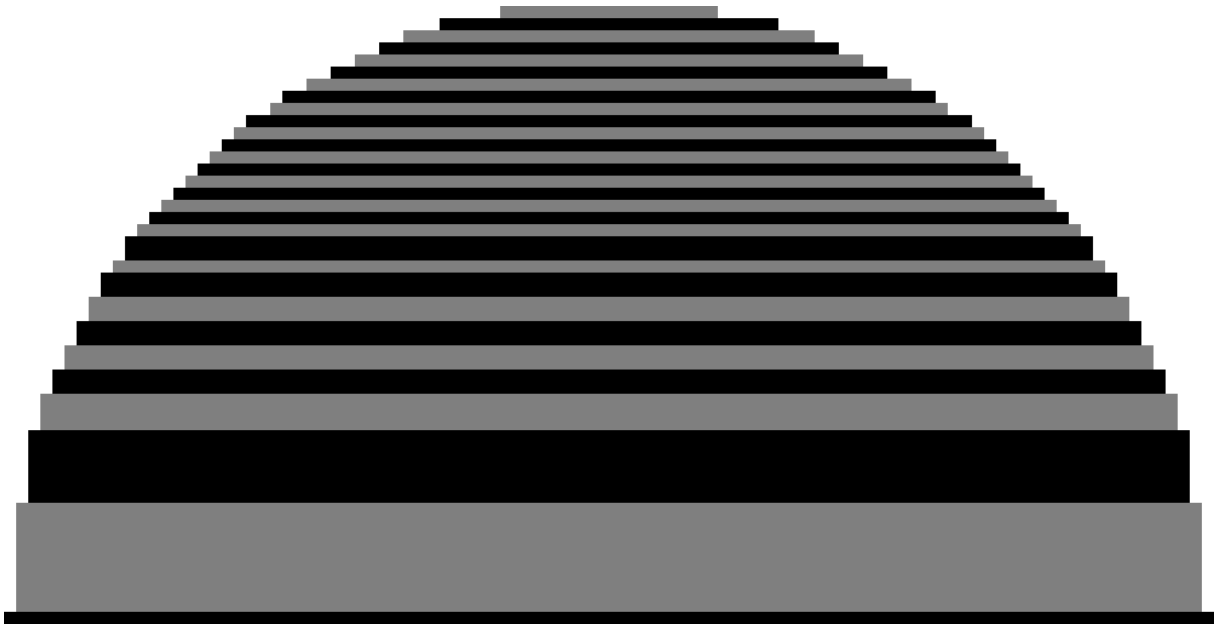
W grach artyleryjskich wybuchy niszczą wycinek terenu znajdujący się w danej odległości od nich. Tak więc problem sprowadza się do wycięcia koła o danym promieniu z danej pozycji siatki. Pierwszym i najprostszym podejściem jest użycie równania koła i zmiana koloru pojedynczych pikseli w siatce przy pomocy funkcji `SetPixel(int x, int y, Color color)`. Niestety to podejście nie sprawdzi się dobrze przy częstym modyfikowaniu maski, ze względu na stosunkowo dużą ilość czasu potrzebną do zmiany wszystkich pixeli, co sugerowane jest w samej dokumentacji [21]. Przy aplikowaniu 100 wybuchów o średnicy 100 pikseli wydajność mierzona w edytorze spadała do poziomu kilkunastu klatek na sekundę.

Zgodnie z sugestią zawartą w dokumentacji wykorzystamy, więc funkcję `SetPixels` modyfikującą na raz cały prostokąt. W tym wypadku musimy jednak opracować metodę umożliwiającą tworzenie koła przy pomocy stosunkowo małej liczby prostokątów.

W tym celu opracowana została funkcja `ListOfRectanglesInCircle` przyjmująca na wejściu promień koła, a na wyjście zawierająca listę par reprezentujących szerokość i wysokość prostokątów tworzących połowę tego koła. Funkcja ta sprawdza odległość rogu prostokąta od środka koła będącego punktem, który nie zawiera żadnych pikseli. Za początkową szerokość na jaką ma być oddalony róg przyjmowany jest promień podany w argumencie, a początkową wysokość 0. Jeśli odległość jest mniejsza niż promień koła róg przesuwany jest o 1 wyżej, aż do momentu kiedy warunek nie będzie spełniony. W tym momencie następuje dodanie prostokąta o odpowiednich wymiarach do listy w przypadku, gdy jego wysokość jest większa od 0 i zmniejszenie badanej szerokości o 1. Prostokąty znajdowane są do momentu kiedy łączna

znalezionych prostokątów jest mniejsza niż promień koła. Należy zauważyć, że ponieważ jako argument przyjmowany jest promień w postaci liczby całkowitej koło zawsze będzie symetryczne względem osi przechodzącej przez środek, a sam algorytm wyznacza prostokąty znajdujące się w jednej połowie koła. Kod implementacji algorytmu został zaprezentowany na listingu 3.1.

Wyniki przedstawionego algorytmu dla ułatwienia przedstawione w postaci wizualnej, gdzie naprzemiennie szare i czarne prostokąty rysowane są podstawie danych zwróconych w postaci listy zostały przedstawione na rysunku 3.2.



Rys. 3.2. Obraz przedstawiający działanie algorytmu wyliczającego listę prostokątów wchodzących w skład koła o promieniu 50 pikseli. Poszczególne prostokąty zostały zaznaczone naprzemiennie na szaro i czarno w celu uzyskania lepszej czytelności.

Użycie rozwiązania bazującego na wycinaniu koła przy pomocy prostokątów okazało się znacznie szybsze. W testach przeprowadzonych w edytorze, w których z bitmapy wycinane było 100 kół o średnicy 100 pikseli, liczba klatek generowanych na sekundę wynosiła w granicach 150–200, co jest wynikiem około dziesięciokrotnie lepszym niż przy przestawianiu koloru pojedynczych pikseli.

3.0.4. Wykrywanie kolizji z terenem

Jedną z ważniejszych rzeczy do zaimplementowania, w grze bazującej na własnym silniku fizycznym są kolizje, mówiące nam czy dany obiekt koliduje z innym. W prezentowanej w pracy grze kolizje opierają się na punktach i dla każdego punktu kolizji do wyboru możliwe są dwie opcje jej wykrywania: kolizja dyskretna sprawdzająca czy obiekt w danej klatce koliduje z podłożem oraz kolizja liniowa, która ponadto wylicza miejsce gdzie nastąpił pierwszy kontakt z otoczeniem. Kolizje dyskretne mogą być odpowiednim wyborem, w momencie , gdy wiemy że prędkość z jaką będzie poruszał się obiekt jest stosunkowo mała - nie przekracza 1 fizycznego piksela na klatkę. W takim wypadku będą one zwracały dokładnie takie same wyniki jak kolizje ciągłe zajmując przy tym mniej czasu na obliczenia. Przy większych prędkościach jednak podejście do nie zdaje egzaminu. Łatwo możemy sobie wyobrazić sytuację kiedy obiekt przesuną się o

```
private List<int2> ListOfRectanglesInCircle(int radius)
        {
            List<int2> rectangles = new List<int2>();
            float currentWidth = radius;
            float currentHeight = 0;
            float heightSum = 0;
            while (heightSum < radius)
            {
                if (radius * radius >= currentWidth * currentWidth + heightSum *
                ↪  heightSum)
                {
                    currentHeight += 1;
                    heightSum += 1;
                }
                else
                {
                    if (currentHeight > 0)
                    {
                        rectangles.Add(new int2((int)currentWidth * 2,
                        ↪  (int)currentHeight));
                    }
                    currentHeight = 0;
                    currentWidth -= 1;
                }

            }
            if (currentHeight > 0)
            {
                rectangles.Add(new int2((int)currentWidth * 2,
                ↪  (int)currentHeight));
            }
            return rectangles;
        }
```

Listing 3.1: Implementacja algorytmu wyznaczającego prostokąty wchodzące w skład koła opracowana w języku C# (opr. wł.)

kiedy szybko poruszający się obiekt przebędzie więcej pikseli niż szerokości ma ściana. W takim wypadku, zaprezentowanym na rysunku 3.3, kolizje dyskretne nie wykryją zderzenia.

Rys. 3.3. Obraz przedstawiający sytuacje kiedy kolizja nie zostanie wykryta przez dyskretne sprawdzanie kolizji. Kolorem czerwonym zaznaczona została pozycja obiektu przed przemieszczeniem, kolorem zielonym pozycję obiektu po ruchu, a kolorem czarnym teren, z którym kolizja nie została wykryta.

W celu rozwiązania tego problemu opracowana została metoda, która pobiera wszystkie piksele znajdujące się od pozycji, w której aktualnie znajduje się obiekt, do pozycji, w której powinien znaleźć się po ruchu. Następnie po kolei zaczynając od miejsca początkowego sprawdzane jest czy dany piksel wskazuje na obecność terenu. Jeśli tak jest miejsce to zapisywane jest jako pozycja kolizji i funkcja jest przerywana, w przeciwnym wypadku funkcja jest kontynuowana. Jeśli kontakt z podłożem nie został wykryty zostaje zwrócona informacja o braku kolizji. Na rysunku 3.4 zobrazowane jest jak przy pomocy kolizji liniowych w sytuacji analogicznej do prezentowanej na rysunku 3.3 zostanie wykryta kolizja.

3.0.5. Poruszanie się po terenie

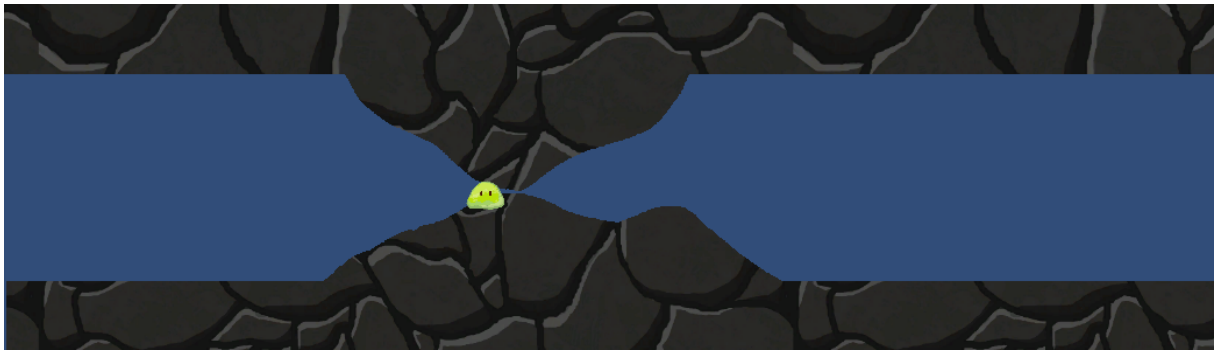
W przypadku poruszania się postaci po zniszczalnym terenie nie możemy opierać się wyłącznie na kolizjach ponieważ nawet niewielkie wzniesienia terenu uniemożliwiałyby dalsze poruszanie się w wyznaczonym kierunku. W związku z tym w systemie odpowiadającym za poruszanie się po terenie musimy uwzględnić nie tylko piksele znajdujące się pomiędzy obecną pozycją gracza, a pozycją docelową, ale także pewien obszar pikseli reprezentowany przez prostokąt. Prostokąt ten ma szerokość odpowiadającą odległości pomiędzy obecną a przewidywaną pozycją gracza, wysokość jest zależna od wysokości gracza i wysokości na jaką chcemy pozwolić mu się wspinać.

Zaimplementowany w grze system umożliwiający na poruszanie się po terenie działa następująco: Dla każdej szerokości od punktu początkowego do punktu końcowego sprawdzane jest, gdzie znajduje wyrwa w pikselach reprezentujących obecność terenu o wysokości na tyle dużej,

Rys. 3.4. Obraz przedstawiający sytuacje kiedy kolizja zostanie zarejestrowana prze użycie kolizji liniowych w sytuacji kiedy dyskretne wykrywanie kolizji nie wykryje kolizji. Kolorem czerwonym zaznaczona została pozycja obiektu przed przemieszczeniem, kolorem zielonym pozycję obiektu po ruchu, a kolorem czarnym teren, z którym kolizja została wykryta. Kolorem jasnoniebieski pokazana jest linia sprawdzonych pikseli, a ciemno-niebieskim miejsce wykrycia kolizji.

aby zmieścił się tam gracz. Jeśli wystarczająco dużo przerwa zostanie znaleziona, sprawdzane jest czy dolna krawędź tej przerwy ma współrzędne w osi y nie większe niż pozycja punktu początkowe w tej osi powiększona o wartość mówiącą jak strome zbocza może pokonywać obiekt. Jeśli obydwa te warunki zostaną spełnione obiekt przesuwany jest do dolnej krawędzi znalezionej przerwy i sprawdzany jest następny rząd. W wyniku takiego przesunięcia gracz może poruszać się po np. wzgórzach co zaprezentowane jest na rysunku 3.5. W przeciwnym wypadku pozycja nie zmienia się i wychodzimy z funkcji. Uniemożliwia to graczowi wchodzenie w za wąskie szczeliny co zaprezentowane jest na rysunku 3.6 oraz wspinaniu się pod zbyt stronę wzgórza, co pokazane jest na rysunku 3.7. Po wykonaniu całej funkcji jeśli obiekt nie został przesunięty w ogóle sprawdzane jest dodatkowo, czy cała sprawdzana przestrzeń nie jest wolna, jeśli jest oznacza, że gracz przestaje kolidować z terenem i należy przesunąć go zgodnie z wyliczeniami systemu odpowiadającego bezpośrednio za ruch w przestrzeni. Przypadek taki zaprezentowano na rysunku 3.8.

Rys. 3.5. Obraz pokazujący gracza poruszającego się po nierównym terenie.



Rys. 3.6. Obraz pokazujący gracza, którego możliwość dalszego poruszania się w prawo została zablokowana przez za wąskie przejście.

Rys. 3.7. Obraz pokazujący gracza, którego dalsza możliwość poruszania się w prawo została zablokowana przez zbyt strome wzgórze.

Rys. 3.8. Obraz pokazujący gracza spadającego po tym jak przestał kolidować z terenem poprzez zsunięcie się poza jego krawędź.

4. TESTY

Testowanie jest niezwykle ważnym aspektem tworzenia oprogramowania. Pozwala ono nie tylko wyłapywać błędy działania programu, czy zweryfikować czy spełnione są wymagania postawione w trakcie projektowania, lecz także weryfikować poprawność przyjętych założeń, czy nawet całych koncepcji stojących za projektem.

W tym rozdziale przedstawione trzy rodzaje testów przeprowadzone na grze powstałej w wyniki pracy inżynierskiej:

1. testy w trybie play mode - testy pozwalające na sprawdzenie poprawności implementacji kodu w środowisku `Unity`,
2. testy wydajnościowe - przeprowadzane za zbudowanej wersji gry, pozwalające sprawdzić czy gra chodzi wystarczająco płynnie i ładuje się wystarczająco szybko na różnych konfiguracjach sprzętowych,
3. testy z użytkownika - testy na zbudowanej wersji gry pozwalające na przetestowanie gotowego projektu na potencjalnych nabywcach w celu uzyskania ich opinii i uwag na jego temat.

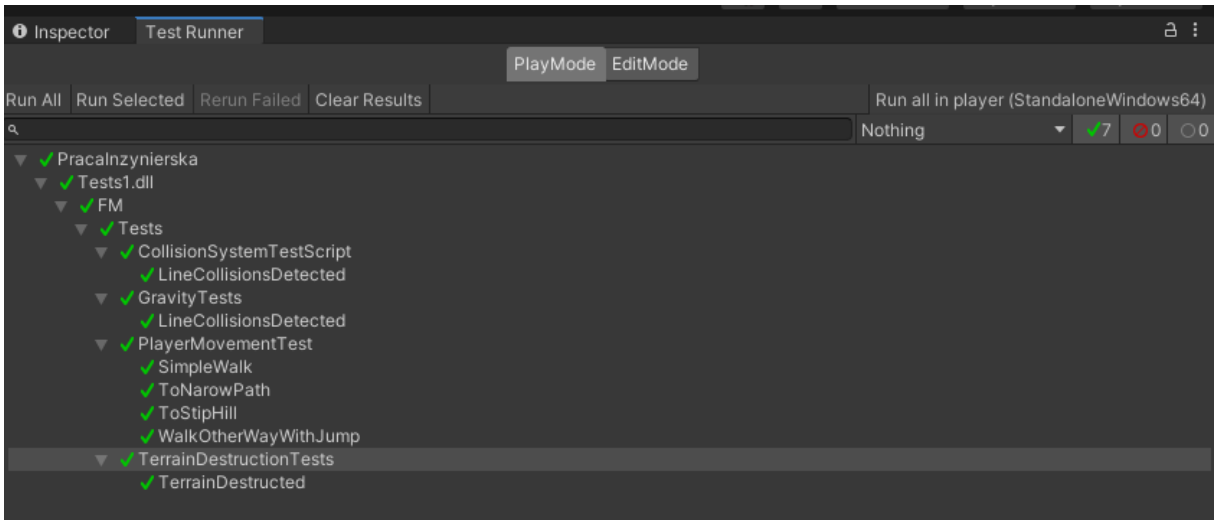
4.1. TESTY W PLAY MODE

Testowanie jest niezwykle ważne w każdej dziedzinie programowania i dobrą praktyką jest jak największe pokrycie kodu testami, na co zwraca uwagę między innymi Robert C. Martin w swojej książce *Clean Code: A Handbook of Agile Software Craftsmanship* [24]. Niestety przy pisaniu gier na silniku zewnętrzny takim jak Unity testowanie jest utrudnione, ponieważ stopień integracji kodu z silnikiem jest na tyle duży, że uniemożliwia to pisanie prostych testów jednostkowych. Możliwe jest jednak wykonanie testów w "play mode", które można zaimplementować w silniku `Unity`. Test taki automatycznie uruchamia wybraną scenę, wykonuje zaplanowaną sekwencję testową, a następnie sprawdza czy otrzymane wyniki są zgodne z oczekiwaniami. Testy takie zostały napisane dla wszystkich głównych systemów (systemy odpowiadające za wykrywanie kolizji, ruch gracza, ruch obiektów w powietrzu, grawitację i destrukcję otoczenia). Wyniki działania testów pokazane są na rysunku 4.1.

4.2. TESTY WYDAJNOŚCIOWE

W celu przetestowania wydajności gry na różnych platformach sprzętowych została przygotowana specjalna scena testowa, na której aktywowane są wszystkie zaimplementowane systemy. Sekwencja testowa trwa określoną liczbę klatek, a czas między kolejnymi klatkami jest mierzony. Maksymalna liczba klatek została ograniczona z poziomu silnika Unity do 60 klatek na sekundę - najwyższej częstotliwości odświeżania możliwej do uzyskania na monitorach na testowanych konfiguracjach. Do przetestowania aplikacji posłużono się 4 różnymi komputerami.

1. PC1: Komputer stacjonarny wyposażony w procesor Ryzen 9 (12 rdzeni, 24 wątki, prędkość rdzenia 3,8GHz do 4.6 GHz w trybie turbo), kartę graficzną RTX 2070 (8 GB pamięci własnej) oraz 16 GB pamięci RAM DDR4.



Rys. 4.1. Obraz prezentujący widok zakładki tests w edytorze Unity po poprawnym przejściu wszystkich testów w playmode.

2. PC2: Komputer stacjonarny wyposażony w procesor Intel Core i5 (4 rdzenie, 4 wątki, prędkość rdzenia 3,8 GHz), kartę graficzną GTX 1060 (6 GB pamięci własnej) oraz 24 GB pamięci RAM DDR4
3. Laptop1: Laptop wyposażony w procesor i5-7300HQ (4 rdzenie, 4 wątki, prędkość rdzenia 2.5GHz do 3.5GHz w trybie turbo) kartę graficzną GTX 1050 (4 GB pamięci własnej) oraz 8GB pamięci RAM DDR4.
4. Tablet1: Kiano Intelect x2: tablet wyposażony w system `Windows 10`, procesor Item Atom x5-Z8300 (4 rdzenie 1,44GHz), 4 GB pamięci RAM DDR3 i wbudowaną kartę graficzną.

Na każdej platformie test został przeprowadzony 10 razy. W tabeli 4.1 przedstawiono uśredniony czas pomiędzy renderowaniem kolejnych klatek oraz średnią liczbę klatek na sekundę uzyskaną przez daną konfigurację sprzętową.

Z danych testów wydajnościowych jasno wynika, że zaimplementowana gra spełnia wymaganie dotyczące płynności rozgrywki nawet na sprzęcie o specyfikacji znacznie gorszej niż zakładana. Utrzymuje ona płynność na poziomie 60 klatek stanowiących górną blokadę nałożoną na grę na wszystkich systemach za wyjątkiem tabletu, gdzie dalej działa płynnie generując średnio 55 klatek na sekundę.

Wnioskować możemy, że w przypadku stworzenia wersji mobilnej telefony, które posiadają lepszą lub nawet porównywalną specyfikację z tabletem, na którym testowana była gra powinny być w stanie zapewnić płynną rozgrywkę.

Poza pomiarami czasów występujących pomiędzy klatkami przeprowadzono również pomiary czasu ładowania sceny testowej, których uśrednione wyniki przedstawione są w tabeli 4.2.

Widzimy, że wszystkie testowane konfiguracje testowe ładowały grę w średnim czasie nie przekraczającym sekundy. Czas ten jest znacząco mniejszy niż zakładane 15 sekund, które stanowią według artykułu [14] czas jaki użytkownik jest w stanie czekać na wczytanie danych.

4.3. TESTY Z UŻYTKOWNIKAMI

Testy z użytkownikami są kluczowym elementem przy implementacji minimalnej wartościowej gry. Gra w takiej formie jest tworzona w celu jak najszybszej możliwości przetestowania

Komputer	Średni czas wykonania klatki	Średnia liczba klatek na sekundę
PC1	16,663ms	60
PC2	16,664ms	60
Laptop1	16,667ms	60
Tablet1	18,042ms	55

Tabela 4.1. Tabela przedstawiająca wyniki wydajności różnych komputerach w zaimplementowanej grze.

Komputer	Średni czas ładowania sceny
PC1	200ms
PC2	194ms
Laptop1	257ms
Tablet1	934ms

Tabela 4.2. Tabela średni czas ładowania sceny testowej na różnych konfiguracjach testowych.

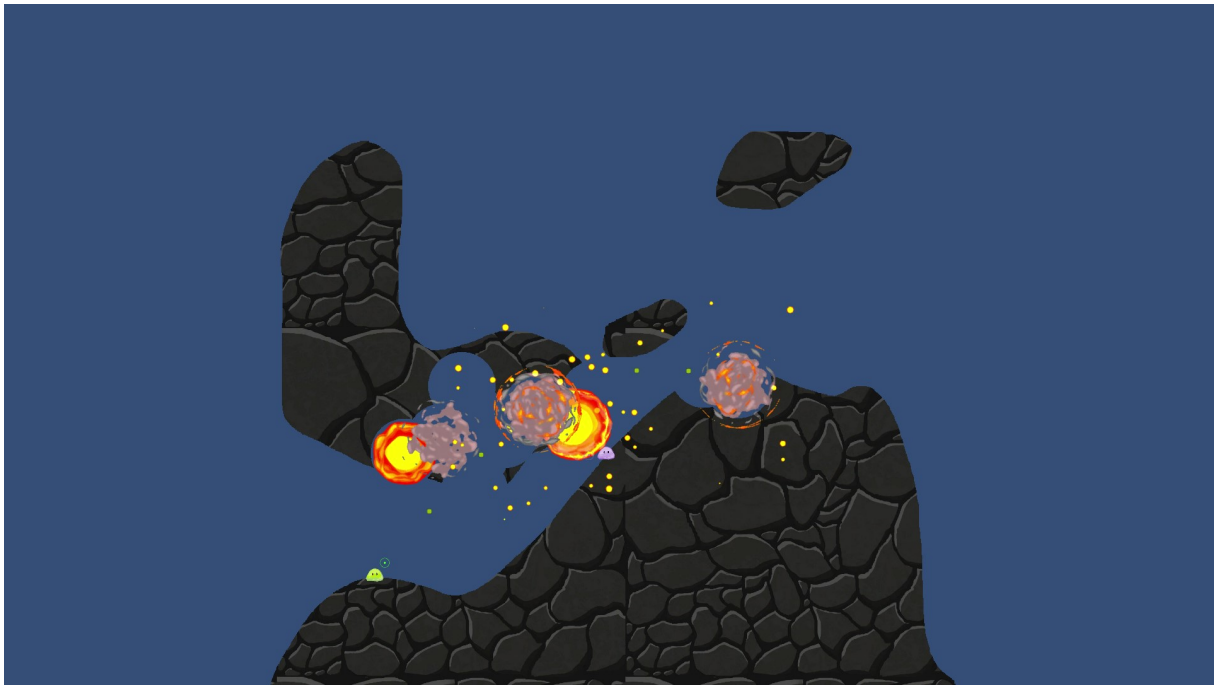
tworzonego produktu z użytkownikami, a uzyskane z takich badań wyniki powinny być podstawowymi przesłankami wpływającymi na kierunek dalszego rozwoju gry. W związku z panującą pandemią koronawirusa przeprowadzenie pełnowymiarowych testów gry wieloosobowej wymagających od graczy przebywania w jednym pomieszczeniu jest w zasadzie nie możliwe. Z tego powodu testy zostały ograniczone w skali, a gra przetestowana została na niewielkich grupach osób, które przez podejmowaną pracę zawodową znajdowały się w jednym pomieszczeniu. Testy zostały przeprowadzone na kilku grupach użytkowników przy pomocy zbudowanej wersji gry, która wraz z instrukcjami instalacji i zrzutami ekranu z rozgrywki udostępniona została na stronie `https://filipmaczko.itch.io/blobs`. Polegały na zorganizowaniu miniturnieju gry, w którym brały udział wszystkie osoby z danej grupy. Zrzuty ekranu prezentujące ciekawe fragmenty rozgrywki prezentowane są na rysunkach 4.2 4.3 4.4.

Po przeprowadzeniu testów uczestniczące w nich osoby pytane były o wrażenia dotyczące gry jak i możliwe usprawnienia oraz największe wady produkcji. Większość osób testujących produkcję wskazała, że gra im się podobała oraz, że byliby oni skłonni nabyć ją, gdy ta zostanie już ukończona.

Jako możliwe usprawnienia gry najczęściej wskazywane były:

1. wprowadzenie interpolacji do systemu sterowania,
2. dodanie dodatkowych map,
3. dodanie dodatkowych broni,
4. zwiększenie wielkości celownika gracza.

Rys. 4.2. Zrzut ekranu pokazujący eliminacje gracza poprzez wybuch w zaimplementowanej grze.



Rys. 4.3. Zrzut ekranu pokazujący pokazujący wiele eksplozji zachodzących w krótkim czasie.

Rys. 4.4. Obraz pokazujący gracza skaczącego w celu uniknięcia kolizji z pociskami przeciwnika.

5. PERSPEKTYWY DALSZEGO ROZWOJU

Obecnie zaimplementowana wersja gry stanowi minimalną wartościową wersję produktu. Jedynie w przypadku dalszego rozwoju przedstawiona gra może mieć możliwość zaistnienia jako komercyjny produkt. W rozdziale przedstawiono możliwości rozwoju projektu wraz z opisem możliwych korzyści wynikających z ich wprowadzenia. Realizacja przedstawionych sugestii pomoże nie tylko doprowadzić grę do stanu, kiedy będzie ona gotowa do wydania, ale również pozwoli wprowadzić ulepszenia do wydanego już produktu.

5.1. ROZBUDOWANIE ROZGRYWKI

W pracy przedstawione zostały testy przeprowadzone z potencjalnymi użytkownikami. Zostały one przeprowadzone w ograniczonych ze względu na panujące okoliczności skalę. W ramach możliwości należałoby poszerzyć skalę przeprowadzonych testów, a co za tym idzie zwiększyć zarówno jakość jak i ilość płynących z nich wniosków.

Już teraz z odpowiedzi udzielanych przez użytkownika możemy wywnioskować, że gra potrzebuje więcej zawartości takiej jak większej ilości map czy większej różnorodności broni. Użytkownicy wskazują także na problemy z czytelnością niektórych elementów takich jak np. celownik.

Zmiany te należy wprowadzić, ponieważ to właśnie one wraz z poprawkami w sferze audio-wizualnej pozwolą na doprowadzenie produktu do wersji możliwej do komercyjnego wydania.

5.2. POPRAWA OPRAWY AUDIOWIZUALNEJ ORAZ GRYWALNOŚCI GRY

Poza zmianami wynikającymi z testów przed wydaniem gry należy także poprawić jej oprawę graficzną oraz stworzyć warstwę audio produkcji. Są to niezwykle ważne aspekty, które w przedstawionej grze zostały zrealizowanie jedynie w stopniu minimalnym pozwalającym przetestować główną mechanikę gry, zgodnie z założeniami tworzenia minimalnej wartościowej gry. Bez nich jednak gra, niezależnie od poziomu jaki prezentuje sama jej mechanika nie ma prawa zaistnieć jako produkt komercyjny. Dzieje się tak, ponieważ gracze do wyboru mają naprawdę wiele pozycji. Według `Statista.com` [25] na samy serwisie `Steam` zostało wydanych 8290 gier. W związku z tym przez pierwsze kilka sekund gra musi przyciągnąć uwagę potencjalnego nabywcy, co nie jest możliwe bez dobrze zrealizowanej warstwy audio-wizualnej.

5.3. PRZENIESIENIE PROJEKTU DO DOTS

Przygotowana w ramach pracy gra została stworzona zgodnie z paradygmatem `ECS`, co powinno znacząco ułatwić proces przenoszenia jej do nowego stosu technologicznego, kiedy ten będzie już ukończony - aktualnie znajduje się on w wersji alpha i niektóre kluczowe dla projektu funkcje nie są jeszcze możliwe do wykorzystania. Przeniesienie projektu do nowe stosu technologicznego powinno dodatkowo poprawić wydajność kodu oraz ułatwić dodawanie bardziej wymagających obliczeniowo funkcjonalności w przyszłości ze względu na możliwość łatwego zrównoleglenia zadań w `DOTS`.

5.4. DODANIE TRYBU SIECIOWEGO

Tryb sieciowy umożliwiający jednoczesną rozgrywkę wielu graczy przez internet pozwoli na granie w przygotowaną produkcję większej ilości osób eliminując potrzebę przebywania w jednym pomieszczenie, by móc wspólnie grać, co jest szczególnie ważne w panujących obecnie czasach.

5.5. STWORZENIE GRY NA URZĄDZENIA MOBILNE

Umożliwienie rozgrywki na telefonie czy tablecie, może dodatkowo poszerzyć grono odbiorców przygotowanej gry. Należy jednak zwrócić uwagę, że będzie ono możliwe jedynie po dodaniu sieciowego trybu rozgrywki ponieważ lokalna gra wieloosobowa w czasie rzeczywistym jest niemożliwa ze względu na charakterystykę urządzenia.

PODSUMOWANIE

W pracy przedstawiony został proces tworzenia minimalnej wartościowej gry, mającej na celu wypełnić lukę zaobserwowaną w rynku gier wideo.

W pierwszej kolejności przeanalizowane zostały możliwości związane z doborem technologii do tworzonego projektu. Jako docelowa platforma wybrany został system `Windows 10`, a jako silnik `Unity`. Następnie przeanalizowane zostały możliwości techniczne dostępne w silniku i wybrany został model fizyczny, który zostanie użyty w grze – fizyka oparta na reprezentacji każdego widzialnego piksela jako zniszczalnego obiektu znajdującego się na bitmapie.

W ramach pracy został opracowany projekt tworzonej gry. W projekcie został uwzględniony zarówno techniczny aspekt projektu jak i aspekt biznesowy. Aspekt techniczny w postaci projektu architektury systemu opartej na nowym paradygmacie wprowadzanym przez `Unity – ECS` zaprezentowany został przy pomocy diagramów klas i aktywności. W ramach aspektu biznesowego przeanalizowane zostały wieloosobowe gry wykorzystujące zniszczalny teren. Na jej podstawie określone zostały rozwiązania powszechnie stosowana w grach tego typu, które zostały uwzględnione przy implementacji gry. Na bazie raportu ESA [4] zidentyfikowany został użytkownik, dla którego tworzony był produkt – mężczyzna w przedziale wiekowym 18–34 lat. Sporządzony został dokument projektu gry odpowiadający formie minimalnej wartościowej gry przyjętej w zakresie prac oraz postawione zostały wymagania, które powinien spełniać system.

W ramach pracy udało się zaimplementować minimalną wartością wieloosobową grę umożliwiającą destrukcję otoczenia w czasie rzeczywistym. Przedstawione zostały testy, z których wyników jednoznacznie wnioskować można, że założenia stawiane przed produktem zostały spełnione, a w niektórych wypadkach takich jak np. czas ładowania mapy były znacząco lepsze niż oczekiwane.

Przedstawione zostały możliwe formy rozwoju projektu, które powinny umożliwić doprowadzenie go do poziomu jakości pozwalającego na komercyjne wydanie produktu oraz jego dalszy rozwój po komercyjnym wydaniu.

BIBLIOGRAFIA

[1] Tom Wijman. The World's 2.7 Billion Gamers Will Spend 159.3 Billion on Games in 2020; The Market Will Surpass 200 Billion by 2023. https://newzoo.com/insights/articles/newzoo-games-market-numbers-revenues-and-audience-2020-2023/. ost. dost. 26 listopada 2020.

[2] Michael Kelly. How Among Us has become one of the most successful and important games of 2020. https://dotesports.com/streaming/news/how-among-us-has-become-one-of-the-most-successful-and-important-games-of-2020. ost. dost. 26 listopada 2020.

[3] InnerSloth. Meet our team! http://www.innersloth.com/About.php. ost. dost. 26 listopada 2020.

[4] Entertainment Software Association. 2019 Essential Facts About the Computer and Video Game Industry. https://www.theesa.com/wp-content/uploads/2019/05/ESA_Essential_facts_2019_final.pdf. ost. dost. 26 listopada 2020.

[5] Richard Hill-Whittal. *Indie Games: Podręczniek niezaleznego twórcy gier*. Merlin Publishing, Warszawa, wydanie 1 edition, 2017.

[6] Gry Online. Seria Worms - seria gier. https://www.gry-online.pl/gry-z-serii-i-podobne.asp?ID=390. ost. dost. 26 listopada 2020.

[7] Tyler York. Making Lean Startup Tactics Work for Games. https://www.gamasutra.com/view/feature/168647/making_lean_startup_tactics_work_.php?print=1. ost. dost. 26 listopada 2020.

[8] GameDesigning.org. The Top 10 Video Game Engines. https://www.gamedesigning.org/career/video-game-engines/. ost. dost. 26 listopada 2020.

[9] Wikipedia Contributors. Scorched Earth (video game). https://en.wikipedia.org/wiki/Scorched_Earth_(video_game). ost. dost. 26 listopada 2020.

[10] Wikipedia Contributors. Liero. https://en.wikipedia.org/wiki/Liero. ost. dost. 26 listopada 2020.

[11] Wikipedia Contributors. Worms (series). https://en.wikipedia.org/wiki/Worms_(series). ost. dost. 26 listopada 2020.

[12] PCGamesN. What's the best Worms game on PC? https://www.pcgamesn.com/best-worms-game. ost. dost. 26 listopada 2020.

[13] Marek Grochowski. Analiza wymagań. https://www.is.umk.pl/~grochu/wiki/doku.php?id=zajecia:ppz:analiza_wymagan. ost. dost. 26 listopada 2020.

[14] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), page 267–277, New York, NY, USA, 1968. Association for Computing Machinery.

[15] Make School. IOS GAMES SUMMER ACADEMY 2017. https://www.makeschool.com/academy/track/build-ios-games/game-design-documents/game-design-documents. ost. dost. 26 listopada 2020.

[16] Unity Inc. Entity Component System. https://docs.unity3d.com/Packages/com.unity.entities@0.16/manual/index.html. ost. dost. 26 listopada 2020.

[17] Iwona Rostkowska. Diagram klas. http://zasoby.open.agh.edu.pl/~09sbfraczek/diagram-klas%2C1%2C11.html. ost. dost. 26 listopada 2020.

[18] Beata Frączek. Diagram aktywności. https://zwinnaanaliza.pl/?p=323. ost. dost. 26 listopada 2020.

[19] Rebeca Costa. Beginners' guide to wireframing UIs: benefits & best practices. https://www.justinmind.com/blog/why-wireframing-your-interface/. ost. dost. 26 listopada 2020.
[20] Unity Inc. Introduction to Sprite Masks. https://learn.unity.com/tutorial/introduction-to-sprite-masks-19-3#. ost. dost. 26 listopada 2020.
[21] Unity Inc. Unity documentation. https://docs.unity3d.com/2020.1/Documentation/ScriptReference/. ost. dost. 26 listopada 2020.
[22] Valentin Simonov. 10000 Update() calls. https://blogs.unity3d.com/2015/12/23/1k-update-calls/. ost. dost. 26 listopada 2020.
[23] Mikael Olsson. *C# 7 Quic Syntax Reference. A pocket guide to language, APIs, and library*. Apress, Hammarland, Finland, edycja 2 edition, 2018.
[24] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, wydanie 1 edition, 2008.
[25] Statista.com. Number of games released on steam worldwide from 2004 to 2019. https://www.statista.com/statistics/552623/number-games-released-steam/. ost. dost. 26 listopada 2020.

SPIS RYSUNKÓW

SPIS TABEL